Table of Contents

Lesson 1 Programming Fundamentals (Review)

Programming

A computer **program** is a set of instructions that tell computers what to do. **Software** is the general term given to these instructions. **Programming** is the term given to the process of developing software.

Computer Languages

Computer languages, like human languages, are guided by a precise set of grammatical rules that must be strictly adhered. There are various levels of computer languages. Ultimately all programs are translated into a series of ones and zeros for this is the only format that a computer understands. Despite all their sophistication, a computer basically goes to a location and determines one of two conditions (a one or zero). Primitive programming was done as a series of ones and zeros and was labeled **machine language.** Machine code was very difficult to develop and understand. The sophisticated programs that we know today would not be possible had it not been for the development of **high-level languages**. These languages are geared more for human understanding and logical development. High-level programming languages allow the use of vocabulary that is common to human communication thus making the task of programming easier. Although easier, these programs must be translated into the machine code described above. A **compiler or interpreter** are programs that make those translations.

Types of computer errors

Compilers and interpreters also detect and indicate **grammatical errors** whenever the language is used incorrectly. It is thus very important to learn the vocabulary and syntax rules for a particular language. The complete translation will not take place as long as there are grammatical errors. Once the program is free of such errors the translation will take place and give us an executable code ready to run.

Once we have the executable code, the program is ready to be run. Hopefully it will run correctly and everything will be fine; however that is not always the case. During "run time", we may encounter a second kind of error called a **run time error**. This error occurs when we ask the computer to do something it cannot do. Look at the following sentence:

> You are required to swim from Naples, Italy to New York in five minutes.

Although this statement is grammatically correct, it is asking someone to do the impossible. Just as we cannot break the laws of nature, the computer cannot violate the laws of mathematics and other binding restrictions. Asking the computer to divide by 0 would be an example of a run time error. We would get executable code; however, when the program tries to execute the command to divide by 0, the program will stop with a run time error. Run time errors are usually more challenging to find than syntax errors.

Once we run our program and get neither syntax nor run time errors, are we free to rejoice? Not exactly. Unfortunately, it is now that we may encounter the worst type of error: the dreaded **Logic**

error. Whenever we ask the computer to do something, but mean for it to do something else, we have a logic error. Just as there needs to be a "meeting of the minds" between two people for meaningful communication to take place, there must be precise and clear instructions that generate our intentions to the computer. The computer only does what we ask it to do. It does not read our minds or our intentions! If we ask someone to cut down the tree when we really meant for them to trim the bush, we have a communication problem. They will do what we ask, but what we asked and what we wanted are two different things. The same is true for the computer. Asking the computer to multiply by 3 when we want it to multiply by 5 is an example of a logic error. Logic errors are the most difficult to find and correct because there are no error messages to help us locate the problem. A great deal of programming time is spent on solving logic errors.

Procedural Programming

There are two basic approaches to writing computer programs.

Procedural programming allows the use of memory locations in the computer to be reserved for **variables:** storage locations containing values that can be altered during the course of the program. Such a memory location is given a name that reflects the nature of the data stored there. For example, *hourlyRate* may be the name of a location that holds the amount of pay a person is paid per hour. Values will also have **operations** performed on them. The variable *hourlyRate* and *numOfHours* may have the multiplication operation performed on them to create a new value, *totalPay*. This could be accomplished in a language such as C++ by the following statement:

```
totalPay = hourlyRate * numOfHours;
```

Instructions can be grouped together into logical units called **procedures**. The instructions to input an hourly rate and number of hours worked and to calculate the total pay could be placed in one procedure. Another procedure could determine the Federal withholding tax. **Call** or **invokes** are used to activate these procedures. The following is a simple code that demonstrates these concepts. The code is not a true language code but rather pseudo (artificial) code. It consists of three procedures (methods): *Main*, *FindPay*, *FindFedTax*. The whole purpose of *Main* is to call the other two procedures. *FindPay* is a procedure(method) that inputs the values of pay rate & hours worked and then calculates and displays the total pay. *FindFedTax* determines the federal tax based on a 30% tax rate. Both *FindPay* and *FindFedTax* either return or need the value of totalPay. This is the reason that *totalPay* is placed in parenthesis in the call and title of those procedures. *TotalPay* is an argument of the methods (procedures) . These concepts are dwelt with at greater length in a later lesson.

Example1.1:

```
Main program
  totalPay=FindPay()
  FindFedTax(totalPay);

FindPay()
   Read(hourlyRate);          // This reads hourly Rate into hourlyRate variable location
   Read(numOfHours);          // This reads the number of hours worked into numOfHours variable
```

```
   totalPay = hourlyRate * numOfHours;        // This calculates total pay and stores it at totalPay
   Return totalPay;                  // This returns the value of total pay to the calling method

FindFedTax(totalPay)
     fedTax = totalPay * .30;  // This calculates the federal tax
     Write(totalPay);              // This writes the total pay
     Write(fedTax);                // This writes the federal tax.
```

Object-Oriented Programming

Object-Oriented Programming (OOP) is similar to the procedural method; however it envisions the components of a program as real world objects. In looking at a Windows applications environment, we see icons (pictures) that represent certain actions. For example, a picture of a trashcan may indicate the action of deleting a file. Instead of a complex command to delete the file, the picture is treating this process as an object (trash can).

Objects are made up of states and methods. The **states** or data items of an object are the components that define what the object is. These states describe the object's **attributes**. An automobile object would have attributes such as make, model, year, color, price etc. All automobiles contain these attributes although the specific attributes would be different for different automobiles. One automobile object may have the following specific attributes: make (Dodge), model (Charger), year (2010), color (red), and price ($35,000). Another automobile object may have the following: make (Chevrolet), model (Malabu), year (2009), price (28,000). States are very similar to variables in procedural programming.
Objects also can have **methods** to accomplish a task. Just like procedures in procedural programming may perform a certain action, methods in OOP perform certain actions. An automobile object can move, be washed etc. Methods describe how these actions are accomplished. While states of an object could be called adjectives that describe the components of the object, methods are **verbs** that describe the actions that can be applied to an object. Object-Oriented Programming binds the states (variables) and the methods (procedures) together in one package. This binding of the two is called **encapsulation**.

A method should be so well written that the user is unaware of the details of how the methods are executed. The user must simply understand the **interface** or interaction of the method to the object. For example, it is not necessary for someone to understand how a television remote control works in order to use the remote to change the stations or the volume. The user of the remote could be called a **client** that only knows how to use the remote to accomplish a certain task. The details of how the remote control performs the task are not necessary for the user to use the remote. Likewise an automobile is a complex mechanical machine with a simple interface that allows users without any (or very little) mechanical knowledge to start, drive and use it for a variety of functions. Drivers do not need to know what goes on under the hood. In the same way a user of an object does not have to understand how the objects methods are implemented.

Classes are the definitions from which objects are created. Classes and objects are often confused with one another; however, there is a subtle but important difference best explained by the following example. A plaster of Paris mold consists of the design of a particular figurine. When the plaster is

poured into the mold and hardened, we have the creation of the figurine itself. A class is analogous to the mold, for it holds the definition of an object. The object is analogous to the figurine, for it is an **instance** of the class. Examples and further explanations are given later in the course.

Components of a Program

A program consists of several different types of pieces.

1) **instructions** statements that execute and perform some action. We describe a program as a set of instructions where each **instruction** is an action to be performed.
2) **control structure**—These are statements that control the execution of a block of instructions. Controls the order of how instructions are executed.
3) **function**—this is a group of instructions that computes and returns one value
4) **method--** It is a group of instructions that performs some task. The difference between a method and a function is that a function computes and returns a single value whereas a method can perform any type of task.
5) **expression** is a list of number, operations or other values that are used to determine a single value. Ex. 7+8 hours * rate rate + 1 are examples of expressions. They all boil down to a single value.
6) **comments** are instructions to the reader of a program. They are ignored by the computer but are very important in documenting a program and to help explain the logic used in the code.
7) **algorithm** A plan of attack. A systematic approach to a problem. There are several programming type algorithms. They are used as a guide to write the program and should always be done before programming begins. We will be studying types that follow the top-down or step-wise-refinement process.

Comments in C++

Comments in C++ are formed the same way as they are in Java. Two basic formats for comments are

1. // everything following this until the end of the line is a comment
2. C++ , just like Java could have comments beginning with /* and ending with */. Note: When using /* the end of line does not end the comments. Only */ ends the comments in this case. Generally /* */ are used in the heading of a program while // are generally used in the code section.

Six basic types of instructions

1. assignment statement
2. write statement (get information from the computer to the user)
3. read statement (gets data into the program for processing)
4. call to a procedure (this is a call to a specialized program of several instructions)
5. conditional statement (determines whether a condition is true or false and controls the execution of a set of instructions based on that result)
6. loop (repeat a series of instructions)

From our definitions above we could have defined the last two as control structures that determine which instructions are to be executed; however traditional programming refers to all of them as fundamental instructions.

Review of the 6 fundamental instructions

Memory storage, variables constants and data type

Memory in a computer can be classified with two attributes: name and data type. We explore data types but basically it means the kind of data that can be stored there. We can use a mailbox as an analogy to memory. A group of mailboxes in an office or hotel front desk can be classified by its size which is an indication of the type of mail it can hold. For example: a mailbox that will only hold postcards will be smaller than one designed for letters which also will be smaller than one designed for packages. The type of mail that a mailbox can hold is a characteristic or attribute of the mailbox. Mailboxes in a hotel are usually small to only hold notes (or perhaps keys). Another attribute is the name given to the mailbox which identifies the person to which it belongs.

Memory in a computer is also identified by a name and a data type (the kind of data it can hold which determines its size).

A variable is a memory location (that has a name and an associated data type) whose value can change or be altered during the course of the program.

A variable name in C++ can consist of any combination of letters (upper and lower case), numbers and the underscore _ character. The exception is that a variable name must begin with a letter. NOTE: Do not use a blank space in any name. If you want to separate words use the underscore character _ Example total_yearly_income hours_worked

Programmers have developed a system of naming classes, variables and methods that should be followed (although it is not a grammatical necessity in all languages). These rules are as follows:

1) When naming a variable, method or function, capitalize the first letter of each word AFTER the first word. All others are lower case.
   Examples: grossPay sumOfProducts pay person2 totalYearlyIncome
2) When naming a class (C++ classes are explained later) capitalize the first letter of each word in the name is capitalized and all the others are lower case

A **variable** is defined as a memory location whose value can change during the execution of the program.

A **constant** is defined as a memory location whose value is initialized and NOT changed during the execution of the program. The standard practice of naming constants is to use ALL upper case letters.

In Java the word final preceded the data type and name of the constant. In C++ the word **const** precedes the data type and name of the constant. Examples of this will be shown later.

Review of the assignment statement

The assignment statement in C++ is basically identical to the one you learned in Java. An assignment statement has two components a left and a right side separated by the = operator. The = operator in this case does not mean equal but rather assign the value to.

The left side (to the left of the = operator) contains one and only one variable (memory location with a name that can change the value it contains). The right side contains an expression. An expression is a group of variables, number or numbers or any combination which produces one value.

total = total + 50;

The left side contains only the one variable total. The right side is an expression that takes the current value that is in total and adds 25 to it. The new one value is now assigned to total.

Thus if total contained the value of 100 before this statement then it will contain 150 after this statement is executed.

Other examples of assignment statements.

payRate= 9.25;

grossPAY= 100;

grossPay=payRate * hours;

euros= dollars * eurosPerDollar;

Just as in Java, every instruction in C++ ends with a semicolon ;

Review of the Write statement

A write statement is used to get information out from the computer to the user. It is used to print the result of some operation or to prompt the user for some feedback.

In Java we learned the System.out.print statements as follows.

System.*out*.print ("How many dollars to you want to convert? ");
This is a write statement to let the user know that the program needs data. (How many dollars to convert.

System.out.println(dollars + " dollars => " + eruos + " euros"); This is a write statement that prints the answer to the problem with labels.

The Write statement in C++

The write statement in C++ is the **cout** instruction

The `cout` statement in C++ invokes a sequence of characters that will be displayed to the screen. Just as Java requires certain classes to be imported to a program for use with certain functions, the cout statement can only be used when we include the statement
#include <iostream> at the beginning of the program.

The `cout` statement has the insertion operator << which inserts the string of characters into the output stream that goes to the screen.

Example 1.2 A simple C++ program using the `cout` statement.

```
// This program prints to the screen the words:
// PI = 3.14
// Radius =4
// Circuference = 25.12

#include <iostream>
using namespace std;
const double PI = 3.14;

int main ()
{
        double radius;
        radius = 4.0;

        cout << "PI = " << PI << endl;
        cout << "Radius = " << radius << endl;
        cout << "Circumference = " << 2 * PI * radius << endl;
        return 0;
}
```

Everything in bold (everything above the int main() located on the previous page) is considered the header or global section. Everything else is in the main method.

```
/* This program prints to the screen the words:
PI = 3.14
Radius = 4
Circumference = 25.12
*/
```

The next statement, the #include statement, is a command that directs the compiler to "include" the appropriate library to the program. It is considered a directive.

#include <iostream >

Every program needs other routines attached to it so that it may execute properly. This one is needed so that the cout (write) statements can be used.

using namespace std;

This statement is generally included in most C++ programs. Its meaning is explained later in the course. For now just remember to use it for your programs.

const double PI = 3.14;

This is how C++ declares a constant. It is almost the same as Java except it uses the word **const** instead of final. Note that we use the double data type. In C++ just like Java the data type precedes the name of a variable or constant. We explain data types later but for now we use the double data type which is a floating point data type (it has a fractional component).

Every C++ program has a **main** function which indicates the start of the executable instructions. Every main must begin with a left brace { and end with
a right brace }. The main program in C++ can be either a void or non-void method. In this case it is a non-void method which indicates that it must return something. It will return the integer 1; The main program could also have been written as void main() in which case there would be no return statement, but it is customary to use the non-void type. The main program heading is easier in C++ than it is is Java. It is either

```
int main()                  or   void main()
{                                {

   return 0;
}                                }
```

The variable declarations are identical to those in Java.

The remaining three statements (not including the return statement) are the write statements.
Just as the + sign is used as the delimiter (separator) in the system.out statements of Java the << is used as the insertion operator (and separator) in C++.
cout << "PI = " << PI << endl;
This will first print the literal PI = to the screen. We know it is a literal because just like Java it is enclosed with double quotes. When a word is not enclosed with quotes it will be taken as a variable or constant and only the contents of the variable or constant (not the name) will be printed.
There are exceptions to the above rule. C++ has certain reserved words. The word endl is a command to go to a new line. The statement above will thus print
PI = 3.13 and then go to a new line

NOTE: \n inside quotes is another way to go to the next line

cout<<"HI\n"; // This will print the word HI and then go to the next line.

cout << “Radius = ” << radius << endl;

This statement prints
Radius = 4 (assuming the value in Radius was 4) and then it goes to the next line

cout << “Circumference = ” << 2 * PI * radius << endl;

This statement prinst
Circumference = 25.12

The 25.12 is found from the expression 2 * PI * radius.

Review of the Read statement

The read statement gets a value from the user (in this case from the keyboard) and stores it into a variable.

Just as the cout statement transfers data from the computer to the “outside” world, the cin statement transfers data into the computer from the keyboard.

The cin statement has what is called the extraction operator >>. This serves as a separator between input variables, allowing more than one memory location to be loaded with a single `cin` instruction.

double rate ;
double hours;

cin >> rate >> hours;

This is a read statement that will allow two double values to be entered from the keyboard with the first going into the memory location called rate and the second going to the memory location called hours.

Just as in Java, since these are coming from the keyboard, they should be preceded by a write (cout) statement.

cout << “Please input the hourly pay rate and the number of hours worked” << endl;
cin>> rate >> hours;

or

cout << “Please input the hourly pay rate” << endl;
cin >> rate;
cout << “Please input the number of hours worked” << endl;
cin >> hours;

Recall what the endl command is for. It forces the cursor to the next line.

Example 1.3 The following program uses the read and write instructions.

```
/*          Here is a sample payroll program that calculates gross pay based on hourly pay rate
            and  hours worked.
/*
#include <iostream>
using namespace std;

int main ()

{
        double  payRate, hoursWorked, grossPay;

        cout << "Please input the hourly pay rate" << endl;
        cin >>payRate;
        cout << "Please input the number of hours worked" << endl;
        cin >> hoursWorked;
        grossPay=payRate* hoursWorked;
        cout << "The gross Pay is $" << grossPay;

        return 0;
}
```

Review of the conditional Statement

The conditional statement in C++ is identical to the one in Java.

It is often called the if-else statement.

```
if (totalIncome > 100000)
     fedTax=.35;
else
    fedTax=.20;
```

This is a conditional statement that checks to see if totalIncome is greater than 100000. If it is true then we have an assignment statement that set fedtax to 25% (.25). If it is false then it does not execute the statement under the if but rather the statement after the else. It that case, the fedtax is 20% (.20). Sometimes the conditional statement contains just the if component. In that case we don't do anything special if the condition is false.

```
if (totalIncome > 100000)
     fedTax=.35;
```

In C++ just as it is in Java, if we want more than one instruction to be executed after the if (or after the else in case the condition is false) we must enclose those statements within curly braces (a block).

```
if (totalIncome > 100000)
{
    fedTax=.35;
    stateTax=.40;
}
else
{
    fedTax=.20;
    stateTax=.30;
}
```

We can have nested if statements.

```
if (grade >= 90)
        System.out.println("You get an A!");
else if (grade >= 80)
        System.out.println("You get an B!");
else if (grade >= 70)
        System.out.println("You get an C!");
else if (grade >= 60)
         System.out.println("You get an D!");
else
        System.out.println("You get an F!");
```

Review of the While loop

A loop is a sequence of instructions that can be repeated 0 or multiple times. There are several loop structures in C++ and they are identical to the ones in Java. For now we just review the while loop.

```
while (condition)  {
     inst 1;
     inst 2;
 }
```

This will cause the two instructions to be executed until the condition is false. As long as the condition is true the three instructions inside the braces { } of the while loop will be executed.

Example: We want to find the mean grade of a group of grades read in from the keyboard.

Write a program that will read an undetermined amount of grades from the keyboard and then will calculate the average of those grades and print that average to the screen.

Notice that we don't know how many grades will be read. We want to read each legitimate grade and add that to some total. As we read a grade we will need to keep the number of grades that we have so

that the final total will be used as the divisor of our total grades. [Average grade equals the total sum of all the grades divided by the number of grades read in]

We begin by writing an algorithm. An algorithm is just a plan on how to solve a problem. We study algorithms in more detail in the next lesson.

Algorithm: Grade Average Program

1.Repeat the following sequence until all grades have been read
   a. Read in a grade
   b. Add that grade to total grades
   c. increase the number of grades read by 1
2. Divide total grades by number grades and assign that value to average grade
3. Write average grade

We must repeat instructions until all the grades have been read. A computer cannot by itself determine when the last legitimate grade is read. There must be a flag data (called **sentinel data**) that indicates that all the grades have been read in.

Sentinel or flag data is usually a value that could not possibly be a legitimate value. In this case for example a negative number could not be a legitimate value. If the program tells the user that is inputting the data to type a negative number when there are no more grades, then the computer program will "know" that all the grades have been read in.

The program would only stop when we input some invalid value. In this case, if we put in a -1(or any negative number) then the loop will quit and the program terminate. Such values that are used to indicate some type of termination are called **sentinel values.**
NOTE about while loops that use sentinel data.

If we are reading a value that uses a sentinel data we must first read in a value to test it before entering the loop. In other words we must read a grade and test it to make sure it is not our sentinel data before we add that value to our total grade. That test must be done both before the loop starts and then again at the end of the loop. Such a process is called priming the read.

Here is a revised algorithm of the problem.
Read grade
while (grade != to sentinel)
    gradeTotal←gradeTotal + grade
    numberOfGrades ←numberOfGrades + 1 loop
    read grade
averageGrade← gradeTotal/numberOfGrades
Write averageGrade

Example 1.4 The following is the C++ program. Make sure you understand each instruction.

#include <iostream>

```
using namespace std;
int main ()
{               double grade;
                double totalGrade=0; // totalGrades will accumulate sum of all grades
                double numberOfGrades=0;  //holds the total amount of grades
                cout <<"Enter a grade and type a negative number to quit" << endl;
                cin >> grade;
                while (grade >=0)
               {
                       totalGrade=totalGrade + grade;
                       numberOfGrades=numberOfGrades +1;
                       cout <<"Enter a grade and type a negative number to quit" << endl;
                       cin >> grade;
                }
                if (numberOfGrades !=0)
                {
                  double averageGrade=totalGrade/numberOfGrades;
                  cout <<"The average grade is "<< averageGrade;
                }
                else
                  cout <<"No grades were read in.";

    return 0;
}
```

We learned that there are six basic fundamental instructions in traditional languages: read, write, assign, loop, conditional and call to a method. We have reviewed the first five and their implementation in C++which except for the read and write are identical to Java. The call to a method instruction is an instruction that goes to a set of instructions that perform some action (called a method). This instruction will be reviewed later in the course.

From Java to C++ * instructions the same in both Java and C++

Instructions	**Java**	**C++**
Read	Scanner kdb=new Scanner(System.in) double hours=kbd.nextDouble;	cin >> hours;
Write	System.out.println("hours = "+hours);	cout << "hours = " << hours << endl;
Assignment *	grossPay=hoursWorked * payRate;	
Conditional *	if (hours > 40) grossPay = 40 * payRate + (hoursWorked -40) * payRate * 1.5;	

While Loop*	while (grade >=0)
	{ instructions inside the loop;
	}	

Lab 1 Programming Fundamental Review and working with Visual C++

Purpose of this lab is to give the student experience in writing C++ programs.

Part 1 Visual C++ IDE (Integrated Development Environment)

The first part of this lab will introduce you to the Visual Studio that allows C++ programs to be written, compiled and run from an IDE.

1. Create a Lab 1 folder
2. Click on Visual studio icon when new pop up window appears click on the Microsoft Visual studio icon (same icon) that appears there.
3. Select file→New→Project
4. In the new pop up window select visual C++ →Win 32 Console application
5. In the name section type First Program in the Location type P:\ then browse to your Lab 1folder.
6. Click OK→
7. At next screen click Next
8. Select console application and check empty project box→Click Finish
9. From the top menu select Project→Add new Item→ C++file
10. In the name type Hello→ click Add
11. Then type in the following code in the new window that pops up.

```
#include <iostream>
using namespace std;

int main()
{
        cout << "Hello C++" << endl;
        return 0;
}
```

12. Then select Build→Build Solution (This will compile the program and show you any syntax errors you have)
13. Then select Build→Profile guided optimization→ Run This will run the program and display the results in the console window.

The window should say

Hello C++
Press any key to Exit

Press any key and then print the code to be turned into the instructor.
(File→ Print)
Part 2 Problems

Problem 1.1 Create a new C++ Project called MilesPerHour. Write a program that will input miles traveled and hours spent in travel. The program will determine the miles per hour.
Sample Run:

Please input the miles traveled
375
Please input the hours spent in travel
6
The average miles per hour is 62.5
Press any key to continue . . .

Problem 1.2 Create a new C++ project called PayRoll and write a program that will input yearly hours worked and the payRate per hour. The program will print the grossPay and it will then find netPay based on the following criteria.

If grossPay is less than 20,000 then federal Income tax is 10%
If grossPay is 20,000 <= < 5000 then federal income tax is 15%
If grossPay is 50000 or greater than federal income tax is 30%

The program should print the gross pay, the tax on that income and the net pay (grossPay – tax).

Sample Run:

Please input the hours worked for the year
2050
Please input the payRate per hour
15
The gross pay is $ 30750
The tax is $4612.5
The net pay is $26137.5
Press any key to continue . . .

Sample Run:

Please input the hours worked for the year
2100
Please input the payRate per hour
50
The gross pay is $ 105000
The tax is $31500
The net pay is $73500
Press any key to continue . . .

Lesson 1 Summary

1) **instructions** statements that execute and perform some action. We describe a program as a set of instructions where each **instruction** is an action to be performed.
2) **control structure**—These are statements that control the execution of a block of instructions. Controls the order of how instructions are executed.
3) **function**—this is a group of instructions that computes and returns one value
4) **method**-- It is a group of instructions that performs some task. The difference between a method and a function is that a function computes and returns a single value whereas a method can perform any type of task.
5) **expression** is a list of number, operations or other values that are used to determine a single value. Ex. 7+8 hours * rate rate + 1 are examples of expressions. They all boil down to a single value.
6) **comments** are instructions to the reader of a program. They are ignored by the computer but are very important in documenting a program and to help explain the logic used in the code.
7) **algorithm** A plan of attack. A systematic approach to a problem. There are several programming type algorithms. They are used as a guide to write the program and should always be done before programming begins. We will be studying types that follow the top-down or step-wise-refinement process.

Comments in C++

Comments in C++ are formed the same way as they are in Java. Two basic formats for comments are

1. // everything following this until the end of the line is a comment
2. C++ , just like Java could have comments beginning with /* and ending with */. Note: When using /* the end of line does not end the comments. Only */ ends the comments in this case. Generally /* */ are used in the heading of a program while // are generally used in the code section.

Six basic types of instructions

1. assignment statement
2. write statement (get information from the computer to the user)
3. read statement (gets data into the program for processing)
4. call to a procedure (this is a call to a specialized program of several instructions)
5. conditional statement (determines whether a condition is true or false and controls the execution of a set of instructions based on that result)
6. loop (repeat a series of instructions)

C++ and Java are very similar. In particular the assignment, loop and conditional statements are identical. The big difference is in the way C++ reads data and writes information.

Every C++ program has a **main** function which indicates the start of the executable instructions. The main program in C++ can be either a void or non-void method.

```
int main()                  or   void main()
{                                {


   return 1;
}                                }
```

The read statement in C++ is the **cin >>** statement and the write statement is the **cout<<** statement. The endl statement is one way C++ goes to the next line.

From Java to C++ * instructions the same in both Java and C++

Instructions	**Java**	**C++**
Read	Scanner kdb=new Scanner(System.in) double hours=kbd.nextDouble;	cin >> hours;
Write	System.out.println("hours = "+hours);	cout << "hours = " << hours << endl;

Assignment *

```
grossPay=hoursWorked * payRate;
```

Conditional *

```
if (hours > 40)
    grossPay = 40 * payRate + (hoursWorked -40) * payRate * 1.5;
```

While Loop*

```
while ( grade >=0)
  {   instructions inside the loop;

  }
```

Lesson 2: Algorithms, Data Types and Control Instructions in C++

Algorithms

Before writing a program in a computer language it is important to develop a plan of attack. An algorithm is a method for solving a problem using a finite sequence of instructions. A recipe is an example of an algorithm for preparing food. There are various algorithmic methods used in computer science. This lesson uses the concept of a **solution tree** using a **step-wise- refinement** or **top-down approach**.

There are several steps involved in developing an algorithm for a computer program and they are listed and explained below:

Example 2.1

Steps		Example
1)	State the problem	Write a program to compute the passing average and the number of failing grades for a given class.
2)	Clarify the problem	Clarification: A negative number will indicate the end of data. A passing grade is a grade >= 60
3)	Show the sample input	Sample Input 85 55 95 75 32 -1 sentinel data
4)	Show the sample output	Sample output 85 2
5)	List the input variables- (variables that have to be read in) choose descriptive names for all memory locations	input variable grade
6)	List the output variables (variables that will be printed) choose descriptive names	passAvg numFailing
7)	List other variables. (variables that are needed to help solve the problem but are not read in and not necessarily printed out)	numPassing passingSum
8)	Draw the solution using a step-wise refinement picture. (explained on next page)	

Step 8

Rules of the Tree

1) Maximum 5 branches per node
2) Only one control structure per node
3) Keep the vertical structure
4) No fundamental instruction from the root

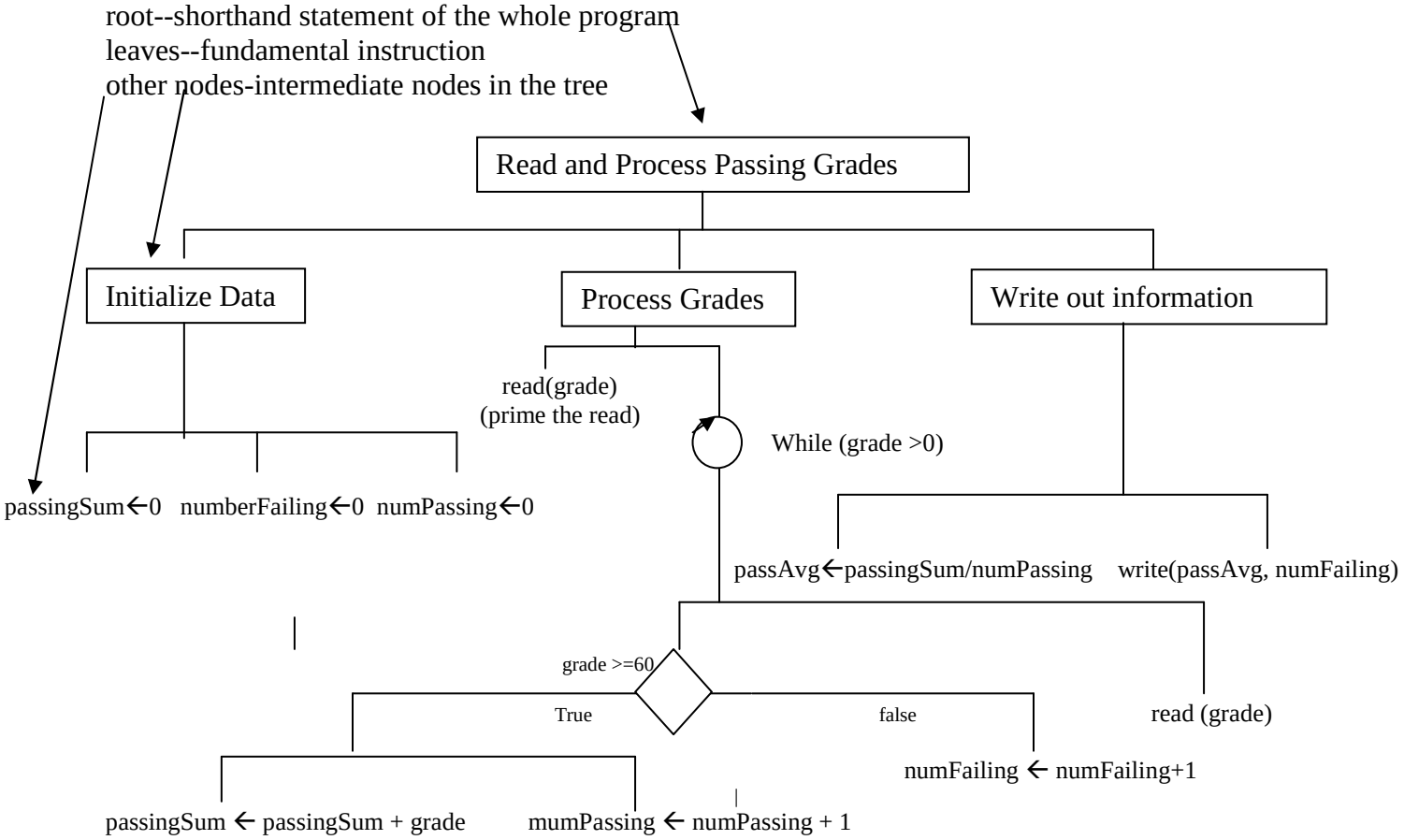
Algorithm symbols used

← assignment statement used in algorithm for the assignment operation =

 Conditional statement used in algorithm for the if and if else statement

 Loop statement used in algorithm for the while for and do while statement

root--shorthand statement of the whole program
leaves--fundamental instruction
other nodes-intermediate nodes in the tree



9) Write the algorithm-found by listing leaves from left to right using appropriate indention of the solution tree from step 8.

Algorithm

```
passingSum<----0
numberPassing<---0
numberFailing<---0
read (grade)              //priming the read
while (grade >0)
      if (grade >=60)
          passingSum<---passingSum + grade
          numberPassing<---numbePassing+1
   else
          numFailing<---numFailing + 1
   read(grade)
passAvg<----passingSum/numberPassing
write(passAvg)
write(numFailing)
```

10) Test the Algorithm using an execution trace

85 55 95 75 32 -1

Set up columns for each of the variables and keep track of their values as the program is executed.

	passingSum	numberPassing	numFailing	grade	passAvg
initial values	0	0	0	X	X
read grade				85	
passing add	85	1			
read				55	
not passing					
add 1 to numFailing			1		
read				95	
passing	180	2			
read				75	
passing	255	3			
read				32	
not passing			2		
read				-1	
Less then 0					
Fall out of loop & solve passAvg	255/3				85
Write passAvg and number Failing	85	2			

Suppose our example data file had the following test grades (all of which are failing grades)
32 18 59 58 49

This is skewed data. But our program divides by number of passing grades which in this case would be 0 so our computer would be asked to divide by 0 which is impossible.

Exercises:

1) Solve the above problem so that it could handle the skewed data

2) Complete steps 5-10 of the algorithm development for the following problem:

Statement: A local hog farmer raises hogs and sells each one when its weight exceeds 250 pounds. Given a list of hogs' weights, determine the average weight of the hogs to be sold and average weight of the hogs that need more fattening.

Clarification: A negative weight will be the sentinel data.

Sample Input:
258
221
236
192
252
250
260
-1

Sample Output:
THE AVERAGE WEIGHT OF HOGS TO BE SOLD IS 256
THE AVERAGE WEIGHT OF HOGS TO BE KEPT IS 224
(((((PERFORM AN EXECUTION TRACE FOR YOUR ALGORITHM)))))

Control Structures in C++

Part 1: The For Loop

Just like Java C++ also has a for loop and a do while loop. They are essentially the same in both languages.

The format for the Java for loop is:

for (int count =start; count<= finish; count = count +1) {

}

In parenthesis we have three parts (separated by a ;).

The first part is the initial value that we assign to count. (it is an assignment statement that also serves as declaring the value of count as an integer). It is followed by a ;

The second part is the condition of termination. As long as count <= to finish we will continue. When count is > than finish then the loop will exit. Finish can be any value. It can be a variable that has a value or it could be a literal number.

The third part changes the value of count. Note: count could be any variable, but it must be changed or else the loop would never end. In this case, count is incremented by one each time through the loop so eventually it will become larger than the value in finish. Note: count++ is a shortcut for count=count+1 see Appendix E.

A for loop is used when the program knows the number of times the loop will be executed. It uses a variable to hold the count (the number of times the loop is executed).
Count could be set at any number and the condition could be anything and count could be incremented by more than one, or it could even be decremented!!! The programmer must make sure that whatever condition he or she gives, they must insure that the loop will end at some point.

Suppose the user at the keyboard wants to input a certain number of temperatures and have the computer determine the average temperature. We can develop a program that asks the user how many temperatures there are and then we can develop a for loop that will loop that many times asking for the temperature.

Look at the following code

```
#include <iostream>
Using namespace std;

      /** This program reads in a certain number of temperatures.
       *  The amount of temperatures is read first and then the program
       *  reads the temperatures in a for loop and calculates the
       *  average temperature.
       *  Programmer:  Your Name
       *
       */
int main()
{
```

```
      double totalTemperatures =0;
      double temperature;
      int    numberOfTemp;
      cout<<”Please input the number of temperatures”;
      cin>>numberOfTemp;
      for (int count =1; count <= numberOfTemp; count++)
      {
            cout<<"PLease input a temperature ";
            cin>>temperature
            totalTemperatures += temperature;
      }
      cout<<"The average temperature is "<<  totalTemperatures/numberOfTemp);

   return 0;
}
```

Note: cout++ is a C++ shortcut for cout=cout+1;
totalTemperatures += temperature; is a C++ shortcut for
totalTemperatures=totalTemperatures + temperature;
double is a data type for floating point numbers (those having fractional components)
int is a data type for integers

Sample run:

```
PLease input the number of temperatures
4
PLease input a temperature
79
PLease input a temperature
52
PLease input a temperature
84
PLease input a temperature
37
The average temperature is 63.00
```

Part 2: while loop

We have worked with the while loop but we review it here.

```
while (Condition) {
  Statement(s)
}
```

The condition is a Boolean expression and Statements can be one or more Java statements.

A while loop’s condition is evaluated BEFORE the Statements are performed. It is possible that the loop will not be executed even once.

The while loop is flexible. In fact all for loops can be implemented as a while loop although it is more cumbersome. In general if the program knows how many times a loop should execute (either from input from the user or program calculation) then a for loop should be used.

Example

If a variable numberOfItems (which has been read or calculated) determines the amount of times a loop is executed it can be implemented as a while loop although a for would be better.

As a for loop	as a while loop
	count=1;
for (int count=1; count <=numberOfItems; count++)	
{	while (count <= numberOfItems) {
	Statements
Statements	count++;
}	}

Part 3: do while loop

In addition to the for and while, C++ offers a do while loop. It's format is:

```
do   {
   Statement(s)
}
while (Condition);
```

The big difference between the do and while loop is that the do loop condition is evaluated after the statements in the loop are executed. That means that the loop will always be processed at least once!

Thus the do loop is used when the statements must be repeated at least once.

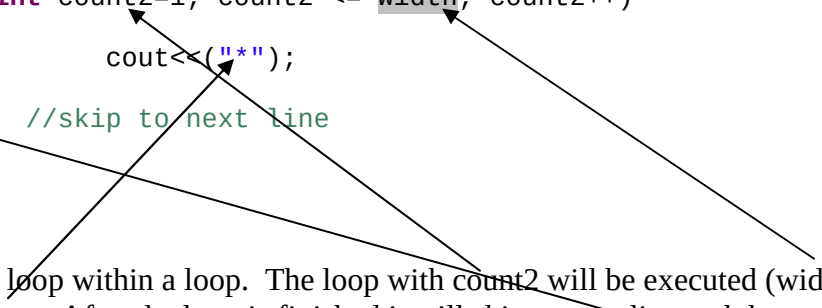
Part 4: Nested loops

Loops can be nested together.

Look at the following code:

```
#include<iostream>
using namespace std;

int main()
{

      int height,width; // the height and width of a box
      cout<<"Please input the height of the box as an integer ";
      cin >> height;
      cout<<"Please input the width of the box as an integer ";
      cin>> width;
      for (int count1 =1; count1 <= height; count1++)
      {
```

```
            for (int count2=1; count2 <= width; count2++)
            {
                        cout<<("*");
            }
      cout<<endl;  //skip to next line

      }
}
```



Notice that we have a loop within a loop. The loop with count2 will be executed (width) times which will place a star in a row. After the loop is finished it will skip to next line and the process starts all over again (This is done height times.).

C++ has 3 types of loops. Which one should you use?

1. If the problem is a counting problem where the number of required repetitions can be determined in advance or by input from the user, then use the for loop. No priming of the read. No use of sentinel data.
2. Otherwise if the problem is such that the loop's statements should be performed at least once, use the do while loop. Do not use priming the read or sentinel data. Can use other means (other than sentinel) to indicate end.
3. Otherwise use the while loop.

Nested conditional statements and the Switch statement

As we learned from earlier lessons the conditional statement (if and if else) can be simple or compound. Any valid C++ statement can be included under the if including another if statement. Although this is often usual it can be confusing at times. If we are dealing with integer or char data types there is another instruction called the Switch statement that we might want to use.

The **switch** Statement

Suppose we wanted to assigned a numeric value to the letter grades A, B,C,D and F.

We could do it with nested if statements
char grade;
int points;
Assuming that grade has been given a char value we can determine the point value as follows

```
if (grade ==’A’)
   points = 4;
else if (grade==’B’)
   points =3;
else if (grade ==’C’)
  points =2;
else if (grade==’D’)
  points=1;
```

```
else
  points=0;
```

We could do this with the switch statement

```
switch (grade)
{
        case ‘A’:
                points=4;
                break;
        case ‘B’:
                points=3;
                break;
        case ‘C’:
                points=2;
                break;
        case ‘D’:
                points=1;
                break;
        default:
                points=0;
                break
}
```

The general format is

```
Switch (variable)      // the variable that is used in each condition
{
        case  value :   // where value is an actual value of the data type of the variable
                 instructions to do if this is the case
                break;   // break is necessary because it gets us out of the switch statement
        case value2:
                instructions;
                break;

        etc.

        default :
                instructions;
                break;
}
```

Data Types in C++

The data types in C++ are similar to the ones in Java.

Data items are the physical symbols that represent information. In C++ each data item must be of a specific type. The data type determines how the data item will be represented in the computer and what kinds of processing the computer will be able to perform on them.

The programmer must tell the computer what form the data will take; however there are several predefined data types in C++.

Different data types will take up different amounts of memory locations in the computer. Ex. Mail boxes can be of different sizes, small ones only can handle letters while larger ones can handle large packages.

1) **float** & **double** data type--this corresponds to a real numbers with fractional components.
   Does 3.0 = 3 ? In math yes but in computer science the answer is NO.
   3.0 consists of both an integer and fractional component. Numbers such 3.0 0.78 3.45 etc. will take up a larger amount of memory than numbers that are just integers. 3 is stored drastically different than 3.0. . Double types can use most all the arithmetic operators such as +,- *, /. Note: * is the multiplication symbol on a computer.

2) **short**, **int** & **long** (Integer)-- These represent integer numbers (-77, 0, 3 etc. NOTE: int is often used to represent a count of items since a count is always a whole number. We can use the arithmetic operators including / but the answer will always be an integer. The operator % is used to find the remainder integer of longhand division. Examples of / and % applied to int (integers)
   / will give the quotient integer of division while % gives the remainder integer of division.

   3 / 15 = 0 15 / 3 = 5 4 / 0 is undefined
   16 / 3 = 5 0 / 4 = 0

   3 % 5 = 3 4 % 5 = 4 5 % 5 = 0
   6 % 5 = 1 7 % 5 = 2 8 % 5 = 3
   15 % 0 is undefined

   (7/2) * 2 + (7%2) = 3 * 2 + 1 = 7

```
int numberInt;
float numberFloat;

numberInt = 5;
numberInt = numberInt / 2;
numberInt = numberInt % 2;
numberFloat = 5;
numberFloat = numberFloat/2;
```

3) **char**-a single character value. Alphanumeric character. ASCII code is a code that describes the workable characters. Note: '8' is not numeric but rather character. It can not be added. Characters can be compared to one another according to their position in the ASCII collating sequence. A is considered less than B because A has a smaller ASCII equivalent number than B.

4) **string** A group of characters. Note: "A" is different than 'A'. "A" string because it is enclosed with double quotes. 'A' is a char because it is enclosed with single quotes.

5) **bool** a Boolean data type that has only two values (true or false)
Ex. bool ischar;
ischar = true; or ischar = false;

Declaring Variables in a program.

Remember that a variable must have a data type and a name.

The data type of each variable must be specified in a C++ program. This means that the programmer must give a name to a memory location and a data type associated with it. This causes the computer to reserve a memory location for that variable by its name and the size of that location will depend on the data type that it is associated with.

Example:

```
float   miles, kms;
bool ischar;
char    letter_1;
int     count;

string message;
```

The above five statements do the following:

1) The computer reserves a memory location for the variable **miles** and another location for the variable **kms**. These two locations are large because they will contain real numbers with fractional components.

2) The computer will reserve a memory location for the variable **letter_1** which will be small since it will only hold one char.

3) A memory location called **count** is reserved for integers.

4) A group of memory locations is automatically reserved for a group of characters called **message** since message is defined as a string. This is not really a primitive data type and when we use string as such we must include the following in the heading. #include <string>

5) a memory location called **ischar** is reserved for the values true or false.

The following is a list of the pre-defined data types (what we might call primitive data types) in C++[1]

`char`	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
`short int (short)`	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`int`	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`long int (long)`	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean value. It can take one of two values: true or false.	1byte	`true` or `false`
`float`	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
`double`	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`long double`	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`wchar_t`	Wide character.	2 *or* 4 bytes	1 wide character

Notice that string is missing because string is really a combination of several character data types. Whenever we use a string data type we must include the following at the beginning of the program
#include<cstring>

More on string data type will be explained later in the course.

Data Conversions

Rules
1. Whenever an integer and a floating point variable or constant are mixed in an operation, the integer is changed temporarily to its equivalent floating point.
2. The automatic conversion of rule 1 is called **Implicit TYPE COERCION.**
3. Whenever a floating point is coerced into an integer, loss of information occurs and in C++ this value is truncated.
4. Type changes can be made explicit by placing the value to be changed in parentheses and placing the name of the new type before it. This is called **type casting** or **type conversion**.

EXAMPLE:
intvalue=10.66; float to int. This is type coercion. Implicit
intvalue = int(10.66) This is type conversion (type casting). Explicit.

The result of intvalue in both cases is 10. Why?

[1] The table was taken from the website http://www.cplusplus.com/doc/tutorial/variables/

Note if 2 integers are divided the result is an integer that is truncated.

1) If hits and at_bats are both type int then average = hits/at_bats would truncate the value and give 0. (unless hits and at_bats are equal).

2) average = float(hits) / at_bats;
Or average = hits / float(at_bats);
Since one of the values is now float the other will have implicit coercion to float.

Output formatting
#include <iomanip> Include this for input output formatting (manipulation)

```
cout << fixed << showpoint << setprecision(2);
```

The fixed causes all the numbers to be shown in fixed point (rather than scientific) notation

The showpoint will show the decimal values even if it is 0
The setprecision(2) will set the decimal value precision to 2

cout << endl << endl << endl << "HI";
cout << "!";
The above cout statements will produce three blanks lines then
HI!

intvalue = 5;
cout <<setw(4) << intvalue;

This prints the contents of intvalue right-justified in four columns. setw is the field width.

realValue=3.14159;
cout << setprecision(3) <<realValue;

This prints 3.142. Set precision shows how many digits to right of decimal point to print. The number rounds at that point.

Arithmetic operators relational operators logical operators and Math Functions
Arithmetic operators + - * /
ex. 5 - 16/2 * 3 + (3+2/2)-5

It can be evaluated to give a unique value. What is that value?

Mathematical Functions
Just like Java, C++ has several mathematical functions. Here are just a few

sqrt(x) square root of value of x
pow(x,y) x^y

fabs(x) takes absolute value of x
floor(x) truncates number ex. 16.89 becomes 16
ceil(x) moves up to the next integer
    Ex. 16.1 becomes 17
To round x floor(x+.5)

They can be used in assignment statements such as
hypotenuse = sqrt(pow(side1,2) + pow (side2,2));

When using these math functions we must import them from a library of routines by using the #include<cmath> in the heading.

Relational Operators
Expressions can be compared to each other with relational & equality operators

> greater than	<= less than or equal
>= greater than or equal	== equal
< less than	!= not equal

A condition is any combination that has a value of true or false.

Logical operators combine two or more relational operations
Logical operators

AND && OR || NOT !

Example if (grade >=80 && grade <90) Both must be true for this condition to be true
if (grade ==’A’ || grade==’B’) If one or both are true then the condition is true
if (!(grade==’F’) True if grade does not contain the value F.

NOTE: with conditional statements never check for equality of 2 real numbers. Not all reals can be represented exactly in binary form.

Example: The following will NOT work.

$s1^2 + s2^2 = h^2$

```
cin >>s1 >> s2 >>h;
if (s1*s1 +s2*s2==h*h)
  cout << " is a right triangle";
else
  cout <<"is not a right triangle";
```

We need to create some small constant that will be a tolerable error of accuracy.

```
const    float TOLERANCE =0.00001;


if (fabs((s1*s1+s2*s2)-h*h)) < TOLERANCE)
        cout << "This is a right triangle";
```

Memory locations are given names to identify their locations or address and then given values that fit into the data type that can be placed in that location.

Reserved Words are words that have special meaning in C and cannot be used for other purposes. See Appendix C.

Lab 2: Data Types and Control Instructions in C++

The purpose of the lab is to give the student practice in working with the conditional and loop control structures in C++ and to work with data types and their conversions.

Make sure that you create a Lab 2 folder in your P drive and get into Visual C++. Make sure you understand how to create a project and add C++ files to a project.

Problem 2.1 Working with the do while and switch statement

Bring in the program dowhile.cpp from the Lab 2 folder in the K drive. The code is shown below:

```
// This program displays a hot beverage menu and prompts the user to
// make a selection. A switch statement determines which item the user
// has chosen. A do-while loop repeats until the user selects item E
// from the menu.
```

// PLACE YOUR NAME HERE

```
#include <iostream>
#include <iomanip>
using namespace std;



int main()
{
```

// Fill in the code to define an integer variable called number,
// a floating point variable called cost,
// and a character variable called beverage

```
      bool validBeverage;

      cout << fixed << showpoint << setprecision(2);


      do
      {
            cout << endl << endl;
            cout << "Hot Beverage Menu" << endl << endl;
            cout << "A: Coffee         $1.00" << endl;
            cout << "B: Tea            $ .75" << endl;
            cout << "C: Hot Chocolate  $1.25" << endl;
            cout << "D: Cappuccino     $2.50" << endl <<endl << endl;

            cout << "Enter the beverage A,B,C, or D you desire" << endl;
            cout << "Enter E to exit the program" << endl << endl;
```

// Fill in the code to read in beverage

```
switch(beverage)
{
case 'a':
case 'A':
case 'b':
case 'B':
case 'c':
case 'C':
case 'd':
case 'D':  validBeverage = true;
               break;
default:   validBeverage = false;
}

if (validBeverage == true)
{
      cout << "How many cups would you like?" << endl;
      // Fill in the code to read in number
}
// Fill in the code to begin a switch statement
// that is controlled by beverage
{
case 'a':
case 'A': cost = number * 1.0;
          cout << "The total cost is $ " << cost << endl;
          break;
// Fill in the code to give the case for tea ( $0.75 a cup)
// Fill in the code to give the case for hot chocolate ($1.25 a cup)
// Fill in the code to give the case for cappuccino ($2.50 a cup)

case 'e':
case 'E': cout << " Please come again" << endl;
          break;
default:cout << // Fill in the code to write a message
                // indicating an invalid selection.
            cout << " Try again please" << endl;
}


} // Fill in the code to finish the do-while statement with the
  // condition that beverage does not equal E or e.

// Fill in the appropriate return statement
}
```

Exercise 1: Fill in the indicated code (in bold) to complete the above program. Then compile and run the program several times with various inputs. Try all the possible relevant cased and record your results.

Exercise 2: What do you think will happen if you do not enter A,B,C,D, or E? Try running the program and inputting another letter.

Exercise 3: Replace the line

if (validBeverage == true)

with the line
if (validBeverage)
and run the program again. Are there any differences in the execution of the program? Why or why not?

Problem 2.2 Working with the for loop

Bring in program for.cpp for the Lab 2 folder. This program has the user input a number n and then finds the mean of the first n positive integers. The code is shown below.

```
// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;


int main()
{
      int value;
      int total = 0;
      int number;
      float mean;

      cout << "Please enter a positive integer" << endl;
      cin >> value;

      if (value > 0)
      {
            for (number = 1; number <= value; number++)
        {
                total = total + number;
            } // curly braces are optional since there is only one statement

        mean = float(total) / value;        // note the use of the typecast
                                            // operator here
            cout << "The mean average of the first " << value
             << " positive integers is " << mean << endl;
    }
    else
        cout << "Invalid input - integer must be positive" << endl;

   return 0;
}
```

Exercise 1: Why is the typecast operator needed to compute the mean in the statement mean = float(total)/value;? What do you think will happen if it is removed? Modify the code and try it. Record what happens. Make sure that you try both even and odd cases. Now put float(total)back in the program.

Exercise 2: What happens if you enter a float such as 2.99 instead of an integer for value? Try it and record the results.

Exercise 3: Modify the code so that it computes the mean of the consecutive

positive integers *n, n+1, n+2, . . . , m*, where the user chooses *n* and *m*. For example, if the user picks 3 and 9, then the program should find the mean of 3, 4, 5, 6, 7, 8, and 9, which is 6.

Problem 2.3 Nested Loops

Bring in program nested.cpp from the Lab 2 folder. The code is shown below.

```
// This program finds the average time spent programming by a student
// each day over a three day period.

// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

int main()
{
      int numStudents;
      float numHours, total, average;
      int student,day = 0;     // these are the counters for the loops
      const int MAXDAY=3;
      cout << "This program will find the average number of hours a day"
           << " that a student spent programming over a long weekend\n\n";
      cout << "How many students are there ?" << endl << endl;
      cin >> numStudents;

    for( student = 1; student <= numStudents; student++)
    {
            total = 0;
            for(day = 1; day <= MAXDAY; day++)
            {
                  cout << "Please enter the number of hours worked by student "
                       << student <<" on day " << day << "." << endl;
            cin >> numHours;

            total = total + numHours;
            }

        average = total / MAXDAY;

            cout << endl;
            cout << "The average number of hours per day spent programming by "
                 << "student " << student << " is " << average
                   << endl << endl << endl;
      }

    return 0;
}
```

Exercise 1: Note that the inner loop of this program is always executed exactly three times—once for each day of the long weekend. Modify the code so that the inner loop iterates *n* times, where *n* is a positive integer input by

the user. In other words, let the user decide how many days to consider just as they choose how many students to consider.

Sample Run:

This program will find the average number of hours a day that a student spent programming over a long weekend
How many students are there?
2
Enter the number of days in the long weekend
2
Please enter the number of hours worked by student 1 on day 1
4
Please enter the number of hours worked by student 1 on day 2
6
The average number of hours per day spent programming by student 1 is 5
Please enter the number of hours worked by student 2 on day 1
9
Please enter the number of hours worked by student 2 on day 2
13
The average number of hours per day spent programming by student 2 is 11

Exercise 2: Modify the program from Exercise 1 so that it also finds the average number of hours per day that a given student studies biology as well as programming. For each given student include two prompts, one for each subject. Have the program print out which subject the student, on average, spent the most time on.

Student Generated Code Assignments

Student Generated Problem Option 2.4

Option 2.4: Write a program that performs a survey tally on beverages. The program should prompt for the next person until a sentinel value of –1 is entered to terminate the program. Each person participating in the survey should choose their favorite beverage from the following list:

1. Coffee 2. Tea 3. Coke 4. Orange Juice

Sample Run:

Please input the favorite beverage of person #1: Choose 1, 2, 3, or 4 from the above menu or -1 to exit the program
4
Please input the favorite beverage of person #2: Choose 1, 2, 3, or 4 from the above menu or -1 to exit the program
1
Please input the favorite beverage of person #3: Choose 1, 2, 3, or 4 from the above menu or -1 to exit the program
3

Please input the favorite beverage of person #4: Choose 1, 2, 3, or 4 from the above menu or -1 to exit the program
1
Please input the favorite beverage of person #5: Choose 1, 2, 3, or 4 from the above menu or -1 to exit the program
1
Please input the favorite beverage of person #6: Choose 1, 2, 3, or 4 from the above menu or -1 to exit the program
-1
The total number of people surveyed is 5. The results are as follows:
Beverage Number of Votes
****
Coffee 3
Tea 0
Coke 1
Orange Juice 1

Student Generated Problem Option 2.5

Option 2.5: Suppose Dave drops a watermelon off a high bridge and lets it fall until it hits the water. If we neglect air resistance, then the distance *d* in meters fallen by the watermelon after *t* seconds is $d = 0.5 * g * t^2$ where the acceleration of gravity $g = 9.8$ meters/second^2. Write a program that asks the user to input the number of seconds that the watermelon falls and the height *h* of the bridge above the water. The program should then calculate the distance fallen for each second from $t = 0$ until the value of *t* input by the user. If the total distance fallen is greater than the height of the bridge, then the program should tell the user that the distance fallen is not valid.

Sample Run 1:

Please input the time of fall in seconds:
2
Please input the height of the bridge in meters:
100
Time Falling (seconds) Distance Fallen (meters)
****

0	**0**
1	**4.9**
2	**19.6**

Sample Run 2:

Please input the time of fall in seconds:
4
Please input the height of the bridge in meters:
50
Time Falling (seconds) Distance Fallen (meters)
****

0	**0**
1	**4.9**
2	**19.6**
3	**44.1**
4	**78.4**

Warning-Bad Data: The distance fallen exceeds the height of the bridge

Student Generated Problem Option 2.6

Option 2.6*:* Write a program that prompts the user for the number of tellers at Hebron's Savings Bank that worked each of the last three years. For each worker the program should ask for the number of days out sick for each of the last three years. The output should provide the number of tellers and the total number of days missed by all the tellers over the last three years.

Sample Run:

How many tellers worked at Hebron Savings Bank during each of the last three years ?
2
How many days was teller 1 out sick during year 1 ?
5
How many days was teller 1 out sick during year 2 ?
8
How many days was teller 1 out sick during year 3 ?
2
How many days was teller 2 out sick during year 1 ?
1
How many days was teller 2 out sick during year 2 ?
0
How many days was teller 2 out sick during year 3 ?
3
The 2 tellers were out sick for a total of 19 days during the last three years.

Lesson 2 Summary

ALGORITHMS

An algorithm is a method for solving a problem using a finite sequence of instructions. This lesson uses the concept of a **solution tree** using a **step-wise- refinement** or **top-down approach**.

Steps

1) State the problem
2) Clarify the problem
3) Show the sample input
4) Show the sample output
5) List the input variables-
6) List the output
7) List other variables.
8) Draw the solution using a step-wise refinement picture
9) Write the algorithm-found by listing leaves from left to right using appropriate indention of the solution tree from step 8.
   Algorithm symbols used

   ← assignment statement used in algorithm for the assignment operation =

    Conditional statement used in algorithm for the if and if else statement

    Loop statement used in algorithm for the while for and do while statement

10) Test the algorithm using an execution trace on the variables

LOOPS

C++ has 3 types of loops. Which one should you use?

1. **For loop** If the problem is a counting problem where the number of required repetitions can be determined in advance or by input from the user, then use the for loop. No priming of the read. No use of sentinel data.
2. **Do while** Otherwise if the problem is such that the loop's statements should be performed at least once, use the do while loop. Do not use priming the read or sentinel data. Can use other means(other than sentinel) to indicate end. This can be
3. Otherwise use the **while loop**.

Switch Statement

```
Switch (variable)      // the variable that is used in each condition
{
        case  value :   // where value is an actual value of the data type of the variable
                  instructions to do if this is the case
                 break;   // break is necessary because it gets us out of the switch statement
```

```
        case value2:
                instructions;
                break;

        etc.

        default :
                instructions;
                break;
}
```

Data Types in C++

int long (used for integers)
float double (used for real numbers with fractional components)
bool (true or false
char (one character)
string (not a primitive type but rather a collection of character data types)
(must include #include<string> statement)

Data Conversions

Rules
1. Whenever an integer and a floating point variable or constant are mixed in an operation, the integer is changed temporarily to its equivalent floating point.
2. The automatic conversion of rule 1 is called **Implicit TYPE COERCION.**
3. Whenever a floating point is coerced into an integer, loss of information occurs and in C++ this value is truncated.
4. Type changes can be made explicit by placing the value to be changed in parentheses and placing the name of the new type before it. This is called **type casting** or **type conversion**.

Mathematical Functions
sqrt(x) square root of value of x
pow(x,y) x^y
fabs(x) takes absolute value of x
floor(x) truncates number ex. 16.89 becomes 16
ceil(x) moves up to the next integer
Ex. 16.1 becomes 17

To round x floor(x+.5)

Use #include<cmath> in the heading.

Relational Operators
Expressions can be compared to each other with relational & equality operators

> greater than	<= less than or equal
>= greater than or equal	== equal
< less than	!= not equal

Logical operators

AND &&
OR ||
NOT !

Example if (grade >=80 && grade <90) Both must be true for this condition to be true
if (grade ==’A’ || grade==’B’) If one or both are true then the condition is true
if (!(grade==’F’) True if grade does not contain the value F.

Output formatting
#include <iomanip> Include this in the heading for input output formatting (manipulation)

```
cout << fixed << showpoint <<setw(8) <<setprecision(2);
```

The fixed causes all the numbers to be shown in fixed point (rather than scientific) notation

The showpoint will show the decimal values even if it is 0
The setprecision(2) will set the decimal value precision to 2
The setw(8) will set the width of the number to 8 places (this includes the decimal point)

Lesson 3 Functions and Methods

Review of Methods in Java

A **method** is basically a group of instructions that performs some task.
A **function** is also a group of instructions; however the purpose of a function is to calculate one value and return that value to the method that called it.

A **procedure or module** is sometimes used to refer to either a method or function.

We learned that Java had what it called two types of methods: void and non void methods.

A void method in Java is what we have defined simply as a method.
A non-void method in Java is what we have defined as a function.

C++ has both methods and functions, but unlike Java (which calls them both methods) C++ calls them both functions: void function (method) and value returning function (function).

This author prefers to call void functions methods and value returning functions simply functions; however this lesson will on most occasions stick to the C++ terms.

A key element of structured (well organized and documented) programs is their modularity: the breaking of code into small units called methods. The use of modules allows us to divide the program into smaller workable units. This can be reflected in the tree structure. If our program is too large the tree structure in the algorithm would become too wide for the page. The logic becomes too complex. The purpose of a structured program is to divide and conquer the logic into several small pieces each of which could be tested separately. We can break the tree down into sub-trees so that we can understand the logic of the smaller units and thus when taken as a whole we understand the full solution.

The `int main()` section of our program is a function and, up until now, has been the only coded module used in our programs. We also have used predefined functions such as pow and sqrt which are defined in library routines that we "imported" to our program with the `#include <cmath>` directive. We now explore the means of breaking our own code into modules. In fact, the main function should contain little more than "calls" to other functions. We can think of the main function as a contractor who hires sub-contractors to perform certain duties: plumbers to do the plumbing, electricians to do the electrical work, etc. The contractor is in charge of the order in which these sub-contract jobs are issued.

The `int main()`function consists mostly of calls to functions just like the contractor issuing commands to sub-contractors to come and do their jobs. A computer basically does many simple tasks (modules) that, when combined, produce a set of complex operations. How one determines what those separate tasks should be is one of the skills learned in software engineering, the science of developing quality software. A good computer program consists of several tasks, or units of code, called modules or functions.

In simple programs most functions are called, or invoked, by the main function. Calling a function basically means starting the execution of the instructions contained in that module. Sometimes a function may need information "passed" to it to perform designated tasks.

If a function is to find the square root of a number, then it needs that number passed to it by the calling function. Information is passed to or from a function through **parameters**. Parameters are the components of communication between functions. Some functions do very simple tasks such as printing basic output statements to the screen. These may be instructions to the user or just documentation on what the program will do. Such functions are often called parameter-less functions since they do not need anything passed to them by the calling procedure.

Example 3.1: Look at parameter-less functions and functions with parameters

```
#include <iostream>
using namespace std;

void printDescription(); //Function prototype
void callPaycheck(float, int);
int  main()
{
        float payRate;
        int hours;
        cout << "Welcome to the Pay Roll Program" << endl;

        printDescription(); //Call to printDescription function

        cout << endl << "Please input the pay per hour" << endl;
        cin >> payRate;
        cout << endl << "Please input the number of hours worked" << endl;
        cin >> hours;
        cout << endl << endl;

        callPaycheck(payRate,hours); //Call to Paycheck function
        cout << "We hoped you enjoyed this program" << endl;
        return 0;
}

//*******************************************************
//                    printDescription
//
//   Task:    This function prints a program description
//   Data in:  none
//
//*******************************************************


void printDescription() //The function heading
```

```
{
        cout << "************************************************" << endl << endl;
        cout << "This program takes two numbers (pay rate & hours)" << endl;
        cout << "and outputs gross pay " << endl;
        cout << "************************************************" << endl << endl;
}

//******************************************************
//                    callPaycheck
//
//   Task:    This function computes and outputs gross pay
//   Data in:  rate and time
//
//******************************************************

void callPaycheck(float rate, int time)
{
        float gross;
        gross = rate * time;
        cout << "The pay is " << gross << endl;
}
```

The bold sections of the program show the development and call of two functions: one has not parameters and the other has two parameters.

The function **heading** `void printDescription()`consists of the name of the function preceded by the word `void`. The word `void` means that this function will not return a value to the module that called it. The function name is followed by a set of parentheses. Just like the `main` function, all functions begin with a left brace and end with a right brace. In between these braces are the instructions of the function. In this case they consist solely of `cout` statements that tell what the program does.

Notice that this function comes after the main function. How is this function activated? It must be called by either the `main` function or another function in the program. This function is called by `main` with the simple instruction `printDescription();`.

A **call** to a function (method) is one of the six fundamental instructions. Notice that it consists only of the name of the function (not the word `void` preceding it) followed by the set of parentheses and a semicolon. By invoking its name in this way, the function is called at that moment. The program executes the body of instructions found in that function and then returns to the calling function (`main` in this case) where it executes the remaining instructions following the call.

NOTE: Since methods and functions perform some action it is customary to name them as some action or action words. We thus name the function printDescription rather than just description.

The `callPaycheck` function is a bit different in that it has parameters inside the parentheses of the call, the heading and the prototype. Parameters are the components of communication to and from a function and the call to that function. The function `callPaycheck` needs information from the calling routine. In order to find the gross pay it needs the rate per hour and the number of hours worked to be passed to it. The call provides this information by having parameters inside the parentheses of the call `callPaycheck(payRate,hours);`. Both `payRate` and `hours` are called **actual parameters or arguments**. They match in a one-to-one correspondence with the parameters sometimes called **formal parameters** in the function heading which are called `rate` and `time`:

void callPaycheck(float rate, int time)

The parameters in a function heading are called **formal parameters**.
It is important to compare the call with the function heading.

Call	**Function heading**
Paycheck(payRate,hours);	**void Paycheck(float rate, int time)**

1. The call does not have any word preceding the name whereas the function heading has the word void preceding its name.

2. The call must NOT give the data type before its actual parameters whereas the heading MUST give the data type of its formal parameters.

3. Although the formal parameters may have the same name as their corresponding actual parameters, they do not have to be the same. The first actual parameter, `payRate`, is paired with `rate`, the first formal parameter. This means that the value of `payRate` is given to `rate`. The second actual parameter, `hours,` is paired with `time` the second formal parameter and gives time its value. Corresponding (paired) parameters must have the same data type. Notice that `payRate` is defined as `float` in the main function and
thus it can legally match `rate` which is also defined as float in the function heading. `hours` is defined as int so it can be legally matched (paired) with `time` which is defined as `int` in the function heading.

4. The actual parameters or arguments (`payRate` and `hours`) pass their values to their corresponding formal parameters. Whatever value is read into `payRate` in
the `main` function will be given to `rate` in the `callPaycheck` function. This is called **pass by value**. It means that `payRate` and `rate` are two distinct memory locations. Whatever value is in `payRate` at the time of the call will be placed in `rate`'s memory location as its initial value. It should be noted that if function `callPaycheck` were to alter the value of `rate`, it would not affect the value of `payRate` back in the main function. In essence, pass by value is like making a copy of the value in `payRate` and placing it in `rate`. Whatever is done to that copy in `rate` has no effect on the value in `payRate`. Recall that a formal parameter could have the same name as its corresponding actual parameter; however, they would still be two different locations in memory.

How does the computer know which location to go to if there are two variables with the same name? The answer is found in a concept called **scope**. Scope refers to the time and location that a variable or memory location is active in a program. All variables defined in the `main` function become inactive when a function is called and are reactivated when the control returns to `main`. By the same token, all formal parameters and variables defined inside a function are active only during the time the function is executing. What this means is that an actual parameter and its corresponding formal parameter are never active at the same time. Thus there is no confusion as to which memory location to access even if corresponding parameters have the same name. More on scope will be presented in the next lesson.

Unlike Java, C++ has **function prototypes** which define the name of the functions and their parameter types before the function is called.

Prototype	**Function heading**
void Paycheck(float, int);	**void Paycheck(float rate, int time)**

1. The prototype has a semicolon at the end and the heading does not.

2. The prototype lists only the data type of the parameters and not their name. However, the prototype could list both and thus be exactly like the heading except for the semicolon. Some instructors tell students to copy the prototype without the semicolon and paste it to form the function heading.

Let us look at all three parts—prototype, call and heading:

1. The heading MUST have both data type and name for all its **formal parameters**.

2. The prototype must have the data type and could have the name for its **formal parameters**.

3. The call MUST have the name but MUST NOT have the data type for its **actual parameters**.

Pass by Reference

Suppose we wanted the `callPaycheck` function to only compute the gross pay and then pass this value back to the calling function rather than printing it. We would need another parameter, not to get information from the call but to give information back to the call. This particular parameter could not be **passed by value** since any change made in a function to a *pass by value formal parameter* has no effect on its corresponding actual parameter. Instead, this parameter would be **passed by reference**, which means that the calling function will give the called function the location of its actual parameter instead of a copy of the value that is stored in that location. This then allows the called function to go in and change the value of the actual parameter.
Example: Assume that I have a set of lockers each containing a sheet of paper with a number on it. Making a copy of a sheet from a particular locker and giving that sheet to you will ensure that you will not change my original copy. This is pass by value. On the other hand, if I give you a spare key to a particular locker, you could go to that locker and change the number on the sheet of paper located

there. This is pass by reference.

How does the program know whether a parameter is passed by value or by reference? All parameters are passed by value unless they have the character & listed after the data type, which indicates pass by reference.

Example 3.2:

```
#include <iostream>
#include <iomanip>
using namespace std;

//Function prototypes
void printDescription();                    // prototype for a parameter-less function
void callPaycheck(float, int, float&);      // prototype for a function with 3
                                            // parameters.  The first two are passed
                                            // by value.  The third is passed by reference
int main()
{
float payRate;
float grossPay;
float netPay;
int hours;

cout << "Welcome to the Pay Roll Program" << endl;

printDescription();                                 //Call to Description function

cout << "Please input the pay per hour" << endl;
cin >> payRate;
cout << endl << "Please input the number of hours worked" << endl;
cin >> hours;
cout << endl << endl;

callPaycheck(payrate,hours,grosspay);   // Call to the callPaycheck function
netPay = grosspay - (grosspay * .2);
cout << "The net pay is " << netPay << endl;
cout << "We hoped you enjoyed this program" << endl;
return 0;
}


//*******************************************************
//                      printDescription
//
//    Task:    This function prints a program description
```

```
//    Data in:  none
//
//*******************************************************


void printDescription() //The function heading
{
        cout << "************************************************" << endl << endl;
        cout << "This program takes two numbers (pay rate & hours)" << endl;
        cout << "and outputs gross pay " << endl;
        cout << "************************************************" << endl << endl;
}

//*******************************************************
//                    callPaycheck
//
//    Task:    This function computes gross pay
//    Data in:  rate and time
//    Data out:  gross  (alters the corresponding actual parameter
//
//*******************************************************

void callPaycheck(float rate, int time, float & gross)
{
        gross = rate * time;

}
```

Notice that the function `callPaycheck` now has three parameters. The first two, `rate` and `time`, are passed by value while the third has an & after its data type indicating that it is pass by reference. The actual parameter `grossPay` is paired with `gross` since they both are the third parameter in their respective lists. But since this pairing is pass by reference, these two names refer to the SAME memory location. Thus what the function does to its formal parameter `gross` changes the value of `grossPay`. After the `callPaycheck` function finds `gross`, control goes back to the `main` function that has this value in `grossPay`. `main` proceeds to find the net pay, by taking 20% off the gross pay, and printing it. Study this latest revision of the program very carefully.

Pass by reference parameters are usually reserved for functions that will alter more than one parameter. They are generally used for functions that initialize several variables. There use is limited and thus one of the reasons they are not used in Java.

Value Returning Functions

The functions discussed so far are not “true functions” because they do not return a value to the calling function. They are often referred to as methods in computer science jargon. True functions, or value returning functions, are modules that return exactly one value to the calling routine. In C++ they do this with the return statement. This is illustrated by the cubeIt function shown in the next example.

Example 3.3

```
#include <iostream>
using namespace std;

int cubeIt(int x);        // prototype for a user defined function that returns the cube of the value
                          // passed to it
int main()
{
        int x = 2;
        int cube;

        cube = cubeIt(x);         // This is the call to the cubeIt function
        cout << "The cube of " << x << << " is " << cube << endl;
        return 0;
}

//*****************************************************************
//
//
//  task:                This function takes a value and returns its cube
//  data in:             some value x
// data returned:        the cube of x
//
//*****************************************************************



int cubeIt(int x)                 // Notice the function type is int rather than void.
{
        int num;
        num = x * x * x;
        return num;
}
```

The function `cubeIt` receives the value of x, which in this case is 2, and finds its cube which it places in a local variable `num`. It then returns the value stored in `num` to the call `cubeIt(x)`. The value 8 replaces the entire function call and is assigned to `cube`, That is `cube = cubeIt(x)` is replaced with `cube = 8`. It is not actually necessary to place the value to be returned in a local variable before returning it. The entire function could be written as follows:

```
int cubeIt(int x)
{
   return x*x*x;
}
```

For value returning functions we replace the word `void` with the data type of the value that is returned. Since these functions return one value, there should be no effect on any parameters that are passed to it from the call. What this means is that all parameters of this type of function should be pass by value, NOT pass by reference. Nothing in C++ prevents the programmer from using pass by reference in value returning functions; however, they should never be used.

The callPaycheck function of example 3.2 would work better as a value returning function since it only changes one parameter.

The following is a comparison between the implementation of a method (void function) and a value returning function of the call Paycheck function

	Value Returning Function	**Procedure**
PROTOTYPE	float callPaycheck(float rate, int time);	void callPaycheck(float rate, int time, float & gross);
CALL	grossPay=callPaycheck(payRate, hours);	callPaycheck(payRate, hours, grossPay);
HEADING	float callPaycheck(float rate, int time)	void callPaycheck(float rate, int time, float & gross)
BODY	{	{
	return rate * time;	gross = rate * time;
	}	}

Functions can also return a Boolean data type to test whether a certain condition exists (true) or not (false).

Functions in tree structures

A tree structure can indicate methods.

Example 3.4

Statement: Calculate the area of each rectangle and find the average area of all the rectangles.

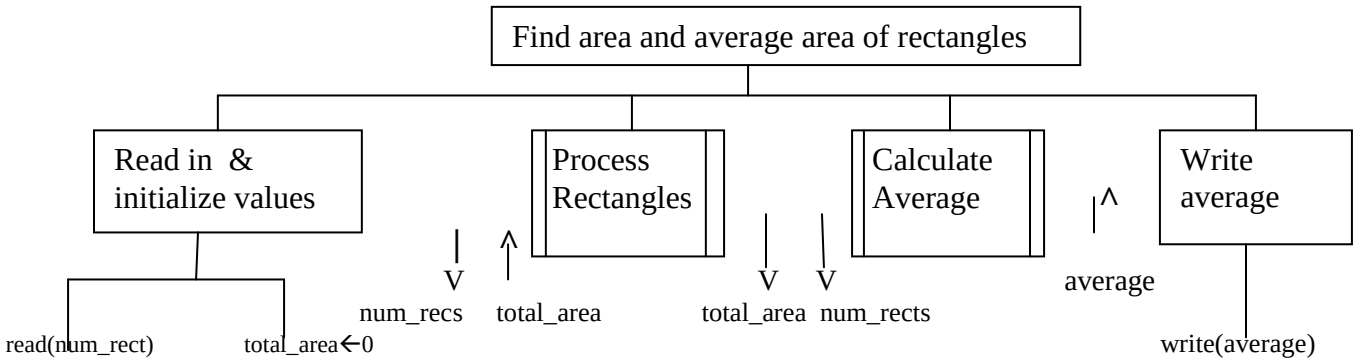
Clarification: The number of rectangles to process will be read in. The program will process the area of each rectangle when given the length and width of each, and keep a running total to find the average area of all the rectangles.

Sample Input: 2 10 5 20 5
# of rectangles L W L W

Sample Output: Area of rectangle 1 10 x 5 is 50
Area of rectangle 2 20 x 5 is 100
Average area of rectangles is 75

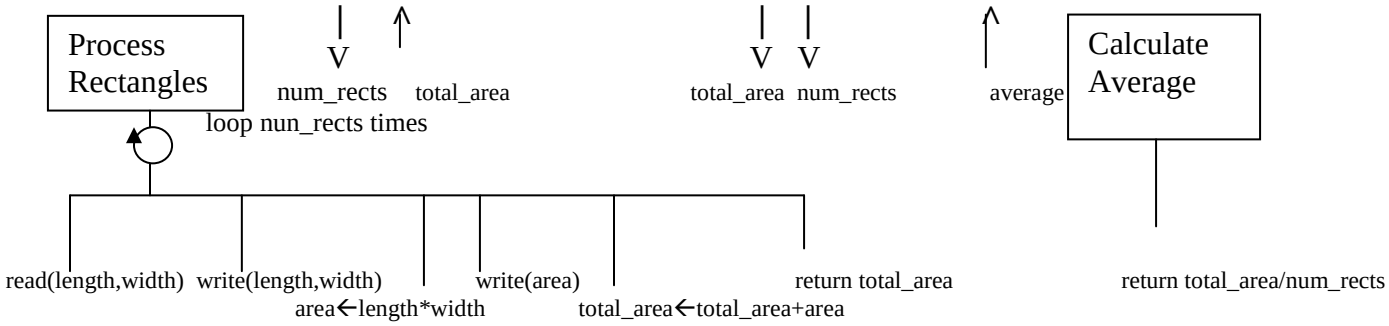
Input Variables:	num_rects	integer	Number of rectangles
	length	integer	Length of each rectangle
	width	integer	Width of each rectangle
Output Variables:	area	integer	Area of each rectangle
	average	real	Average of all rectangles

	length		(Echoed—input is also written out)
	width		(Echoed)
Other Variables:	total_area	integer	sum of all areas



Functions (Methods) are shown in a tree by having double lines on the side. The arrows reflect parameters. Down arrows represent parameter values (pass by value) needed by the function, while up arrows (if only one generally mean a value returning function but could mean a void function with one pass by reference value) indicate values changed or calculated by the function and passed back to the calling method.

Each Function node would then be developed on a separate sheet of paper.



Psudocode for functions can look just like the function call with parameters listed inside a parenthesis. Documentation for functions should follow the example given in this lesson.

Lab 3: Functions and Methods

The purpose of this lab is to give the student practice in working with both void and value returning functions.

Create a Lab 3 folder.

Problem 3.1 Retrieve program newproverb.cpp from the Lab 3 folder. The code is as follows:

```
// This program will allow the user to input from the keyboard
// whether the last word to the following proverb should be party or country:
// "Now is the time for all good men to come to the aid of their _______"
// Inputting a 1 will use the word party. Any other number will use the word country.

// PLACE YOUR NAME HERE

#include <iostream>
#include <string>
using namespace std;

// Fill in the prototype of the function writeProverb.

int main ()
{
        int wordCode;

        cout << "Given the phrase:" << endl;
        cout << "Now is the time for all good men to come to the aid of their ___" << endl;
        cout << "Input a 1 if you want the sentence to be finished with party" << endl;
        cout << "Input any other number for the word country" << endl;
        cout << "Please input your choice now" << endl;
        cin  >> wordCode;
        cout << endl;

         writeProverb(wordCode);

        return 0;
}
```

```
//*********************************************************************
//                      writeProverb
//
//   task:          This function prints a proverb. The function takes a number
//                   from the call. If that number is a 1 it prints "Now is the time
//                   for all good men to come to the aid of their party."
//                   Otherwise, it prints "Now is the time for all good men
//                   to come to the aid of their country"
//   data in:       code for ending word of proverb (integer)
//   data out:      none
//
//   *********************************************************************

void writeProverb (int number)


{
  // Fill in the body of the function to accomplish what is described above
}
```

Exercise 1: Some people know this proverb as “Now is the time for all good men to come to the aid of their country” while others heard it as “Now is the time for all good men to come to the aid of their party.” This program will allow the users to choose which way they want it printed. Fill in the blanks of the program to accomplish what is described in the program comments. What happens if you inadvertently enter a float such as -3.97?

Exercise 2: Change the program so that an input of 1 from the user will print “party” at the end, a 2 will print “country” and any other number will be an invalid choice so that the user will need to repeat their choice.

Sample Run:
Given the phrase:
Now is the time for all good men to come to the aid of their __
Input a 1 if you want the sentence to be finished with party
Input a 2 if you want the sentence to be finished with country
Please input your choice now
4
I'm sorry but that is an incorrect choice; Please input a 1 or 2
2
Now is the time for all good men to come to the aid of their country

Exercise 3: Change the program so the user may input the word to end the phrase. The string holding the user’s input word will be passed to the proverb function instead of passing a number to it. Notice that this change requires you to change the proverb function heading and the prototype as well as the call to the function.
Sample Run:

Given the phrase:
Now is the time for all god men to come to the aid of their _____
Please input the word you would like to have finish the proverb
family
Now is the time for all good men to come to the aid of their family.

Problem 3.2

Retrieve program paycheck.cpp from the Lab 3 folder.
The code is as follows:

```
// This program takes two numbers (pay rate & hours)
// and multiplies them to get grosspay
// it then calculates net pay by subtracting 15%

//PLACE YOUR NAME HERE

#include <iostream>
#include <iomanip>
using namespace std;

//Function prototypes
void printDescription();
void computePaycheck(float, int, float&, float&);

int main()
{
        float payRate;
        float grossPay;
        float netPay;
        int hours;

        cout << setprecision(2) << fixed;

        cout << "Welcome to the Pay Roll Program" << endl;

        printDescription(); //Call to Description function

        cout << "Please input the pay per hour" << endl;
        cin >> payRate;

        cout << endl << "Please input the number of hours worked" << endl;
        cin >> hours;
        cout << endl << endl;
```

```
        computePaycheck(payRate,hours,grossPay,netPay);

        //  Fill in the code to output grosspay

        cout << "The net pay is $" << netPay << endl;

        cout << "We hoped you enjoyed this program" << endl;

        return 0;
}

//   ********************************************************************
//                    printDescription
//
//   task:     This function prints a program description
//   data in:  none
//   data out: none
//
//   ********************************************************************

void printDescription() //The function heading
{
   cout << "************************************************"  << endl << endl;
   cout << "This program takes two numbers (pay rate & hours)" << endl;
   cout << "and  multiplies them to get gross pay "           << endl;
   cout << "it then calculates net pay by subtracting 15%"    << endl;
   cout << "************************************************"  << endl << endl;
}

// *********************************************************************
//                    computePaycheck
//
//   task:          This function takes rate and time and multiples them to
//                   get gross pay and then finds net pay by subtracting 15%.
//   data in:       pay rate and time in hours worked
//   data out:      the gross and net pay
//
//   *********************************************************************

void computePaycheck(float rate, int time, float& gross, float& net)

{
        //  Fill in the code to find grosspay and net pay

}
```

Exercise 1: Fill in the code (places in bold) and change the program so that the function `computePaycheck` has a fourth parameter called net that determines the net pay by subtracting 15 percent from the gross pay. Both `gross` and `net` should be returned to the `main()` function where those values are printed.

Exercise 2: Compile and run your program with the following data and make sure you get the output shown.

Please input the pay per hour
9.50
Please input the number of hours worked
40

The gross pay is $380
The net pay is $323
We hoped you enjoyed this program

Exercise 3: Are the parameters `gross` and `net` in the modified `computePaycheck` function you created in Exercise 1 above, pass by value or pass by reference?

Exercise 4: Alter the program so that `gross` and `net` are printed in the function `computePaycheck` instead of in `main()`. The `main()` function executes only the statement cout << "We hoped you enjoyed this program" << endl; after the return from the function `computePaycheck`.

Exercise 5: Run the program again using the data from Exercise 2. You should get the same results.

Student Generated Code Assignments

Student Generated Code Problem Option 3.3

A C++ program is needed that uses the number of wins and losses a baseball team acquired during a complete season to calculate their percentage of wins. The program should be organized as a main function and three value-returning functions. The first is a parameter-less function that inputs the number of wins and returns this number to the main function. The second is a similar function that does the same thing for the losses. The third function should be passed the wins and looses, which it will use to calculate and return the percentage of wins (number of wins/total_games) to the main function. Main should print the result to two decimal places. Don't forget prototypes!!! Print the code to be turned into the instructor.

NOTE: wins and losses are integer values while percentage is a float. Remember you cannot divide integer by integer and get a float. You will have to convert one of the values to float.

The return statement of the third (value returning function) should look something like this.

return ((float) win/sum); where win is a parameter passed to the function and sum is the addition of win and losses (losses also passed to the function). Also note that the percentage output should be in percent to two decimal places (example: 66.67%) and not just decimal .6667.

Exercise 1: Write the program and run it with the sample run below

Sample Run

Please input the Number of wins: 80
Please input the Number of losses: 40
The percentage of wins is 66.67%

Exercise 2: Redo the problem so that the results are exactly the same, but this time instead of using two value returning functions to input wins and losses, use one void function that will have wins and losses passed by reference. The void function will thus change the value of wins and losses but will not return any value. The sample run should be identical as given in problem two.

Student Generated Code Program Option 3.4

Write a program that will read two floating point numbers (the first read into a variable called `first` and the second read into a variable called `second`) and then calls the function `swap` with the actual parameters (arguments) `first` and `second`. The swap function having formal parameters `number1` and `number2` should swap the value of the two variables.

Sample Run:
Enter the first number
Then hit enter
80
Enter the second number
Then hit enter
70
You input the numbers as 80 and 70.

After swapping, the first number has the value of 70 which was the value of the second number. The second number has the value of 80 which was the value of the first number.

Exercise 1: Compile the program and correct it if necessary until you get no syntax errors.

Exercise 2: Run the program with the sample data above and see if you get the same results.

Exercise 3: The `swap` parameters must be passed by _____. Why?
(Assume that main produces the output)

Option3.5*:* Write a program that will input miles traveled and hours spent in travel. The program will determine miles per hour. This calculation must be done in a function other than `main`; however, `main` will print the calculation. The function will thus have 3 parameters: `miles, hours,` and `milesPerHour.` Which parameter(s) are pass by value and which are (is) passed by reference?

Output is fixed with 2 decimal point precision.

Sample Run:
Please input the miles traveled
475
Please input the hours traveled
8
Your speed is 59.38 miles per hour

Student Generated Code Problem Option 3.6

Option 3.6 : Write a program that will input grades, the number of which is also input by the user. The program will find the sum of those grades and pass it, along with the number of grades, to a function that has a "pass by reference" parameter that will contain the numeric average of those grades as processed by the function. The main function will then determine the letter grade of that average based on a 10-point spread.

90–100	A
80–89	B
70–79	C
60–69	D
0–59	F

Sample Run:

Enter the number of grades
3
Enter a numeric grade between 0-100
90
Enter a numeric grade between 0-100
80
Enter a numeric grade between 0-100
50
The grade is C

Lesson 3 Summary

Just like Java, C++ functions are either void or non void methods. C++ calls them both functions instead of methods. A void function is the C++ name for a Java void method. A value-returning function is the C++ name for a Java non-void method.

The heading for a void function is: **void nameOfFunction()**
Just like Java the word void means that the function does not return any value.
The heading for a value-returning function is : **data type nameOfFunction()**
Just like Java, the data type in the heading is the type of data that will be returned to the calling method. If there are parameters, they are listed inside the parenthesis.

Just like Java, the formal parameters (parameters in the heading of a function) must have the data type before the name of the parameter. Just like Java, the arguments (actual parameters listed in the call to the function) just have their name (no data types).

C++ has what is called **function prototypes** that is basically the same as the function heading. They are located before the main program so that when a reference (call) is made to the function, the compiler is aware that such a function exists. One difference between a function prototype and the function heading is that the function prototype ends with a semicolon while a function heading (just like Java) has a { indicating the beginning of the function. One other difference between a function prototype and a function heading is that the function prototype does not have to include the name of the parameter, just the data types. It is not wrong to include the names and thus the prototype can look exactly like the heading except it has a semi-colon.

```
#include <iostream>
using namespace std;

// Function prototypes
void description();
void paycheck(float, int);     //  could be written   void paycheck (float rate, int time);
```

Pass by Reference

C++, unlike Java, gives the programmer an option of passing primitive data types by either value or reference. When a parameter is passed by reference the called method (function) can alter the value of the actual parameter (argument). The & is used after the data type to indicate pass by reference.

Function Prototype using a pass by reference parameter and its corresponding function heading
void paycheck(float, int, **float&**);
void paycheck (float rate, int time, **float & gross)**

Lesson 4: Scope, Overloading functions, Files, Stubs & Drivers

Scope
The scope of an identifier (variable, constant, function, etc.) is an indication of where it can be accessed in a program. There can be certain portions of a program where a variable or other identifier cannot be accessed for use. Such areas are considered out of the scope for that particular identifier. The header (the portion of the program before main) has often been referred to as the global section. Any identifier defined or declared in this area is said to have **global scope**, meaning it can be accessed at any time during the execution of the program. Any identifier defined outside the bounds of all the functions have global scope. Although most constants and all functions are defined globally, variables should almost **never** be defined in this manner.

Local scope refers to identifiers defined within a **block**. They are active only within the bounds of that particular block. In C++ a block begins with a left brace { and ends with a right brace }. Recall that all functions (including main) must begin and end with a pair of braces which form a block. Other blocks could be defined within blocks and thus the scope of some identifiers would not extend for the entire function.

Variables defined within functions are called **local variables** (as opposed to **global variables** defined in the heading, before main, which have **global scope.)** Local variables can normally be accessed anywhere within the function from the point where they are defined. However, blocks can be defined within other blocks, and the scope of an identifier defined in such an inner block would be limited to that block in which it is defined. A function's formal parameters have the same scope as local variable defined in the outmost block of the function. This means that the scope of a formal parameter is the entire function. The following sample program illustrates some of these scope rules.

Example 4.1

```
#include <iostream>
using namespace std;

const PI = 3.14;

void printHeading();


int main()
{
  float circle;
  cout << "circle has local scope that extends the entire main function" << endl;
    {
          float square;
          cout << "Square has local scope active for only a portion of main" << endl;
          cout << "Both square and circle can be accessed here "
                << "as well as the Global constant PI" << endl;
    }
```

```
  cout << "circle is active here, but square is not" << endl;
  printHeading();
  return 0;
}
void printHeading()
{
   int triangle;
   cout << "The Global constant PI is active here"
        << " as well as the local variable triangle" << endl;
}
```

Notice that the nested braces within the outer braces of `main()` indicate another block in which `square` is defined. `Square` is active only within the bounds of the inner braces while `circle` is active for the entire `main` function. Neither of these are active when the function `Heading` is called. `triangle` is a local variable of the function `printHeading` and is active only when that function is active. `PI`, being a global identifier, is active everywhere.

Formal parameters have the same scope as local variables defined in the outmost block of the function. That means that the scope of formal parameters of a function is the entire function. The question may arise about variables with the same name. For example, could a local variable in the function `printHeading` of the above example have the name `circle`? The answer is yes, but it would be a different memory location than the one defined in the `main` function. There are rules of **name precedence** which determine which memory location is active among a group of two or more variables with the same name. The most recently defined variable has precedence over any other variable with the same name. In the above example, if `circle` had been defined in the `printHeading` function then the memory location assigned with that definition would take precedence over the location defined in `main()` as long as the function `printHeading` was active.

Lifetime is another way of describing the scope of a variable. It refers to the time during a program that an identifier has storage assigned to it.

Scope Rules

1. The scope of a global identifier, any identifier declared or defined outside all functions, is the entire program.

2. Functions are defined globally. That means any function can call any other function at any time.

3. The scope of a local identifier is from the point of its definition to the end of the block in which it is defined. This includes any nested blocks that may be contained within, unless the nested block has a variable defined in it that has the same name.

4. Scope of formal parameters is the same as the scope of local variables of the function.

Why are variables almost never defined globally? Good structured programming assures that all communication between functions will be explicit through the use of parameters. Global variables can be changed by any function. In large projects, where more than one programmer may be working on the same program, global variables are unreliable since their values can be changed by any function or any programmer and are thus not often used. The inadvertent changing of global variables in a particular function can cause unwanted side effects.

Static Local Variables

One of the biggest advantages of a function is the fact that it can be called multiple times to perform its job. This saves much programming time and memory space. The values of local variables do not remain between multiple function calls. What this means is that the value assigned to a local variable of a function is lost once the function is finished executing. If the same function is called again that value will not necessarily be present for that local variable. Local variables start "fresh," in terms of their value, each time the function is called. There may be times when a function needs to retain the value of a variable between calls. This can be done by defining the variable to be a **static.** which means it is initialized at most once and its memory space is retained even after the function in which it is defined has finished executing. Thus the lifetime of a static variable is different than a normal local variable. Static variables are defined by placing the word **`static`** before the data type and name of the variable as shown below.

```
static int totalPay = 0;
static float interestRate;
```

Default Arguments

Actual parameters (parameters used in the call to a function) are often called **`arguments`**. Normally the number of actual arguments must equal the number of formal parameters, and it is good programming practice to use this one-to-one correspondence between actual and formal parameters. It is possible, however, to assign default values to all formal parameters so that the calling instruction does not have to pass values for all the arguments. Although these default values can be expressed in the function heading, they are usually defined in the prototype. Certain actual arguments can be left out; however, if an argument is left out, then all the following arguments must also be left out. For this reason, pass by reference arguments should be placed first (since by their very nature they must be included in the call).

Example 4.2

```
#include <iostream>
#include <iomanip>
using namespace std;

void callNetpay(float& net, int hours = 40, float rate = 6.00);
// function prototype with default arguments specified

int main()
{
```

```
        int hoursworked = 20;
        float payrate = 5.00;
        float pay;   // net pay calculated by the callNetPay function

        cout << setprecision(2) << fixed << showpoint;

        callNetpay(pay); // call to the function with only 1 parameter
        cout << " The net pay is $" << pay << endl;

        return 0;
}

//*************************************************************
//
//                              callNetPay
//
//  taske:      This function takes rate and hours and multiples them to get net pay
//              (no deductions in this pay check!)  It has tow default parameters. If the
//              third argument is missing from the call, 6.00 will be passed as the rate
//              to this function.  If the second and third arguments are missing from the
//              call, 40 will be passed as the hours and 6.00 will be passed as the rate.
//
//  data in:  pay rate and time in hours worked
//  data out: net pay (alters the corresponding actual parameter)
//
//
//*************************************************************

void callNetpay (float& net, int hours, float rate)
{
        net = hours * rate;
}
```

What will happen if `pay is` not listed in the calling instruction? An error would occur that states that the function cannot take 0 arguments. The reason for this is that the `net` formal parameter does not have a default value and so the call must have at least one argument. In general there must be as many actual arguments as formal parameters that do not have default values. Of course some or all default values can be overridden.

The following calls are all legal in the example program. Fill in the values that the call`Netpay` function receives for `hours` and `rate` in each case. Also fill in the value that you expect net pay to have for each call.

callNetpay(pay); The net pay is $_____

callNetpay receives the value of _____ for hours and _____ for rate.

callNetpay(pay, hoursworked); The net pay is $_____
callNetpay receives the value of _____ for hours and _____ for rate.

callNetpay(pay, hoursworked, payrate); The net pay is $_____
callNetpay receives the value of _____ for hours and _____ for rate.

The following are not correct. List what you think causes the error in each case.

callNetpay(pay, payrate);
callNetpay(hoursworked, payrate);
callNetpay(payrate);
callNetpay();

Overloading Functions

Uniqueness of identifier names is a vital concept in programming languages. The convention in C++ is that every variable, function, constant, etc. name needs to be unique. However, there is an exception. Two or more functions may have the same name as long as their parameters differ in quantity or data type. For example, a programmer could have two functions with the same name that do the exact same thing to variables of different data types.

Example: Look at the following prototypes of functions. All having the same name and yet all could be included in the same program because each one differs from all the others either by the number of parameters or the data types of the parameters.

```
int add(int a, int b, int c);

int add(int a, int b);

float add(float a, float b, float c);

float add(float a, float b);
```

Stubs and Drivers

Many IDEs (Integrated Development Environments) have software debuggers which are used to help in the location of logical errors; however, programmers often use the concept of stubs and drivers to test and debug programs that use functions and procedures. A **stub** is nothing more than a dummy function that is called instead of the actual function. It usually does little more than write a message to the screen indicating that it was called with certain arguments. In structured design, the programmer often wants to delay the implementation of certain details until the overall design of the program is complete. The use of stubs makes this possible.

Example 4.3

```
#include <iostream>
```

```
using namespace std;

int findSqrRoo(int x);    //  prototype for a user defined function that returns the square root
                          //   of the number passed to it.
int main()
{
        int number;

        cout << "Please input the number whose square root your want." << endl;
        cout << "Input a -99 when you would like to quit" << endl;
        cin >> number;

        while (number1 != 99)
        {
                cout << "The square root of your number is "<< findSqRoot(number) << endl;
                cout << "Please input the number whose square root your want." << endl;
                cout << "Input a -99 when you would like to quit" << endl;
                cin >> number;

        }
        return 0;
}

int findSrRoot(int x)
{
        cout << "findSqrRoot function was called with " << x << " as its argument\n";
        return 0;
}  //This bold section is the stub.
```

This example shows that the programmer can test the execution of `main` and the call to the function without having the function finding the square root. This allows the programmer to concentrate on one component at a time. Although a stub is not really needed in this simple program, stubs are very useful for larger programs.

A **driver** is a module that tests a function by simply calling it. While one programmer may be working on the `main` function, another programmer could be developing the code for a particular function. In this case the programmer is not so concerned with the calling of the function but rather with the body of the function itself. In such a case a driver (call to the function) could be used just to see if the function performs properly.

Example 4.4

```
#include <iostream>
#include <cmath>
using namespace std.
```

```
int findSqrRoot(int x);    //  prototype for a user defined function that returns the
                           //  square root of the number passed to it
int main()
{
        int number;

        cout << "Calling findSqrRoot function with a 4" << endl;
        cout << "This is the result " << findSqrRoor(4) << endl;
        return 0;
}

int findSqrRoot(int x)
{
      return sqrt(x);
}
```

In this example, the `main` function is used solely as a tool (driver) to call the `findSqrRoot` function to see if it performs properly.

Files

So far all our input has come from the keyboard and our output has gone to the monitor. Input, however, can come from files and output an go to files. To do either of these things we should add the `#include <fstream>` directive in the header to allow files t be created and accessed. A file containing data to be input to the computer should be defined as an `ifstream` data type and an output file should be defined as `ofstream.`

Suppose we have a data file called `grades.dat` that contains three grades and we want to take those grades and output them to an output file that we will call `finalgrade.out`. The following code shows how this can be done in C++:

```
#include <fstream> // This statement is needed to use files
using namespace std;


int main()
{
  float grade1, grade2, grade3; // defines three float variables

  ifstream dataFile;                    // This defines an input file stream.
                                        // dataFile is the "internal" name is used in the
                                        //  program for accessing the data file

  ofstream outFile;                     // This defines an output file stream.
                                        //  outFile is the"internal" name that is used in the
                                        //  program for accessing the output file called
```

```
  outFile << fixed << showpoint;      // These can be used with output files as well as cout

  dataFile.open("grades.dat");         // This ties the internal name, dataFile,
                                       // to the actual file, grades.dat

  outFile.open("finalgrade.out")       //  This ties the internal name, outFile, to
                                       //   the actual file, finalgrade.out.

  dataFile >> grade1 >> grade2         //  The reads the values from the input file
           >> grade3;                  //   into the three variables

  outFile << grade1 << endl;           //   These 3 lines write the values stored in the
  outFile << grade2 << endl;           //    3 variables to the output file
  outFile << grade3 << endl;

  datafile.close();                   //  files should be closed before the program ends.
  outfile.close();

  return 0;
}
```

Reading from a File

Files have an “invisible” end of line marker at the end of each line of the file. Files also have an “invisible” end of file marker at the end of the file. When reading from an input file within a loop, the program must be able to detect that marker as the sentinel data (data that meets the condition to end the loop). There are several ways to do this.

Example 4.5

```
#include <fstream>
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
        ifstream infile;   // defining an input file
        ofstream outfile; // defining an output file

        infile.open("payroll.dat");
        // This statement opens infile as an input file.
        // Whenever infile is used, data from the
        // file payroll.dat will be read.
```

```
        outfile.open("payment.out");
        // This statement opens outfile as an output file.
        // Whenever outfile is used, information will
        // be set to the file payment.out.

int hours;      // The number of hours worked
float payRate; // The rate per hour paid
float net;      // The net pay

if(!infile) // This condition is true if infile(payroll.dat)is not found.
{
        cout << "Error opening file.\n";
        cout << "Perhaps the file is not where indicated.\n";
        return 1;
}
outfile <<fixed) << setprecision(2);
outfile << "Hours      Pay Rate     Net Pay" << endl;

infile >> hours;

while(infile) // as long as data is in the file the loop will continue
{
        infile >> payRate;
        net = hours * payRate;
        outfile << hours << setw(10) << "$ " << setw(6) << payRate
              << setw(5)<< "$ " << setw(7) << net << endl;
        infile >> hours;
}

infile.close();
outfile.close();
return 0;
}
```

This program assumes that a data file exists that contains an undetermined number of records with each record consisting of two data values, `hours` and `payrate`. Suppose that the input data file (`payroll.dat`) contains the following:

```
40 10.00
30 6.70
50 20.00
```

The program will produce the following output file (`payment.out`).

```
Hours    Pay Rate      Net Pay
40       $ 10.00       $ 400.00
30       $   6.70      $ 201.00
```

50 $ 20.00 $1000.00

The input file contains data for one employee on each line. Each time through the `while` loop, information is processed for one employee. The loop executes the same number of times as there are lines (employee records in this case) in the data file. Since there are two items of data for each line (`hours` and `payRate`), these items are read in each time through the loop. Notice that one of the input variables was input before the `while` loop. This is called "priming the read." Input can be thought of as a stream of values taken one at a time. Before the while loop condition can be tested, there has to be something in that "stream." We **prime** the read by reading in at least one variable before the loop. Observe that the statement `infile >> hours;` is given twice in the program: once before the input loop and as the last statement in the loop. The other item, `payrate,` is read in at the very beginning of the loop. This way each variable is read each time through the loop. Also notice that the heading of the output file is printed outside the loop before it is executed.

There are other ways of determining that the end of the file has been reached.
The `eof()` function reports when the end of a file has been encountered. The loop in example 4.5 can be replaced with the following:

```
infile >> hours;

while (!infile.eof())
{
        infile >> payRate;
        net = hours * payRate;
        outfile << hours << setw(10) << "$ " << setw(6) << payRate
              << setw(5)<< "$ " << setw(7) << net << endl;
        infile >> hours;
}
```

In addition to checking to see if a file exists, we can also check to see if it has any data in it. The following code checks first if the file exists and then if it is empty.

```
inData.open("sample2.dat");

if(!inData)
        cout << "file does not exist" << endl;
else if(ch = inData.peek()) == EOF)
        cout << "File is empty" << endl;
else
//rest of program
```

The `peek` function actually looks ahead in the file for the next data item, in this case to determine if it is the end of file marker. `ch` must be declared as `char` data type.

Since the `peek` function looks "ahead" for the next data item, it can be used to test for end of file in reading values from a file within a loop without priming the read.

The following example gives the same result as example 4.5 without priming the read. The portions in bold differ from example 4.5

Example 4.6

```
#include <fstream>
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
        ifstream infile;   // defining an input file
        ofstream outfile; // defining an output file

        infile.open("payroll.dat");
        // This statement opens infile as an input file.
        // Whenever infile is used, data from the
        // file payroll.dat will be read.

        outfile.open("payment.out");
        // This statement opens outfile as an output file.
        // Whenever outfile is used, information will
        // be set to the file payment.out.

int hours;        // The number of hours worked
float payRate; // The rate per hour paid
float net;        // The net pay
char ch;      // ch is used to hold the next value
           // read as character from the file

if(!infile) // This condition is true if infile(payroll.dat)is
           //  not found.
{
        cout << "Error opening file.\n";
        cout << "Perhaps the file is not where indicated.\n";
        return 1;
}
outfile <<fixed) << setprecision(2);
outfile << "Hours      Pay Rate     Net Pay" << endl;

while(ch=infile.peek() != EOF)
{
        infile >> hours;
        infile >> payRate;
```

```
        net = hours * payRate;

        outfile << hours << setw(10) << "$ " << setw(6) << payRate
              << setw(5)<< "$ " << setw(7) << net << endl;
}
infile.close();
outfile.close();
return 0;
}
```

Output Files

Output files are opened the same way as input files: `outfile.open("payment.out")`. Whenever the program writes to `outfile`, the information will be placed in the physical file `payment.out`. The program generates a file stored in the same location as the source file. The user an indicate a different location for the file to be stored by indicating the full path (drive etc.) Remember to close files before ending the program.

Files Used for Both Input and Output

A file can be used for both input and output. The `fstream` data type, which is used for files that can handle both input and output, must have a file access flag as an argument to the open function so that the mode, input or output, can be determined. There are several access flags that are used to indicate the use of the file. The following chart lists frequently used access flags.

Flag mode	Meaning
ios:: in	Input mode. The file is used for "reading" information. If the file does not exist, it will not be created.
ios::out	Output mode. Information is written to the file. If the file already exists, its contents will be deleted.
ios::app	Append mode. If the file exists, its contents are preserved and all output is written to the end of the file. If it dos not exist then the file will be created. Notice how this differs from ios::out.
ios::binary	Binary mode. Information is written to or read from in pure binary format (discussed later in this chapter)

Example 4.7

```
#include <fstream>
using namespace std;

int main()
```

```
{
    fstream test (“grade.dat”, ios::out)
    // test is defined as an fsteam file first used for output
    // ios::out is the file access flag

    // code of the program goes here.  The code will put values in the test file

    test.close();    //  close the file as an output file
    test.open(“grade.dat”, ios::in)
     //  the same file is reopened now as an input file
    //   ios:: in is the file access flag
    //  other code goes here

  test.close();   // close the file
}
```

Lab 4: Scope, Overloading functions, Files, Stubs & Drivers

The purpose of this lab is to give the student practical applications in the concept of scope, function overloading and the use of files.

Create a Lab 4 folder.

Problem 4.1 Retrieve program scope.cpp from the Lab 4 folder. The code is as follows:

```
#include <iostream>
#include <iomanip>
using namespace std;

// This program will demonstrate the scope rules.

// PLACE YOUR NAME HERE

const double PI = 3.14;
const double RATE = 0.25;

void findArea(float, float&);
void findCircumference(float, float&);

int main()
{
        cout << fixed << showpoint << setprecision(2);
        float radius = 12;
        cout <<" Main function outer block << endl;
        cout <<" LIST THE IDENTIFIERS THAT are active here" << endl << endl;
        {
                float area;
                cout << "Main function first inner block" << endl;
                cout << "LIST THE IDENTIFIERS THAT are active here" << endl << endl;

                // Fill in the code to call Area here

                cout << "The radius = " << radius << endl;
                cout << "The area = " << area << endl << endl;
        }
        {
                float radius = 10;
                float circumference;

                cout << "Main function second inner block" << endl;
                cout << "LIST THE IDENTIFIERS THAT are active here" << endl << endl;
```

```
            // Fill in the code to call Circumference here

            cout << "The radius = " << radius << endl;
            cout << "The circumference = " << circumference << endl << endl;
        }

        cout << "Main function after all the calls" << endl;
        cout << "LIST THE IDENTIFIERS THAT are active here" << endl << endl;
        cout << "The radius is now " << radius << endl;
        return 0;
}

//  ****************************************************************
//
//                                  findArea
//
//   task:        This function finds the area of a circle given its radius
//   data in:     radius of a circle
//   data out:    answer (which alters the corresponding actual parameters
//
//****************************************************************

void findArea(float rad, float& answer)
{
        cout << "AREA FUNCTION" << endl << endl;
        cout << "LIST THE IDENTIFIERS THAT are active here" << endl << endl;
        // FILL in the code given that parameter rad contains the radius, that will find
        // the area to be stored in answer
}

//  ****************************************************************
//
//                                  findCircumference
//
//   task:        This function finds the circumference of a circle given its radius
//   data in:     radius of a circle
//   data out:    distance (which alters the corresponding actual parameters
//
//****************************************************************


void findCircumference(float length, float& distance)
{
        cout << "CIRCUMFERENCE FUNCTION" << endl << endl;
        cout << "LIST THE IDENTIFIERS THAT are active here" << endl << endl;
```

// FILL in the code given that parameter length contains the radius, that will find
// the circumference to be stored in distance

}

Exercise 1: Fill in the following chart by listing all the identifiers (function names, variables, constants)

GLOBAL Main Main(inner 1) Main (inner 2) Area Circumference

Global	*Main*	*Main(inner 1)*	*Main(inner 2)*	*Area*	*Circumference*

Exercise 2: For each cout instruction that reads:

cout << " LIST THE IDENTIFIERS THAT are active here" << endl;

Replace the words in all caps by a list of all identifiers active at that location. Change it to have the following form:

cout << "area, radius and PI are active here" << endl;

Exercise 3: For each comment in bold, place the proper code to do what it says.
NOTE: area = Π *r*2
circumference = 2Π*r*

Exercise 4: Before compiling and running the program, write out what you expect the output from this program to be.

What value for `radius` will be passed by `main` (first inner block) to the findArea function?
What value for `radius` will be passed by `main` function (second inner block) to the Circumference function?

Exercise 5: Compile and run your program. Your instructor may ask to see your program run or obtain a hard copy of your work.

Problem 4.2 Retrieve program money.cpp from the Lab 4 folder. The code is as follows

```
#include <iostream>
#include <iomanip>
using namespace std;
```

```
// PLACE YOUR NAME HERE

void normalizeMoney(float& dollars, int cents = 150);

//  This function takes cents as an integer and converts it to do dollars and cents.
//   The default value for cents is 150 which is converted to 1.50 and stored in dollars

int main()
{
        int cents;
        float dollars;

        cout << setprecision(2) << fixed << showpoint;

        cents = 95;
        cout << "\n We will now add 95 cents to our dollar total\n";

        //  Fill in the code to call normalizeMoney to add 95 cents

        cout << "Converting cents to dollars resulted in " << dollars << " dollars\n";

        cout << "\n We will now add 193 cents to our dollar total\n";

        //  Fill in the code to call normalizeMoney to add 193 cents

        cout << "Converting cents to dollars resulted in " << dollars << " dollars\n";

        cout << "\n We will now add the default value to our dollar total\n";

        //  Fill in the code to call normalizeMoney to add the default value of cents

        cout<<"Converting cents to dollars resulted in " << dollars << " dollars\n";

        return 0;
}



//  ************************************************************
//
//                                  normalizeMoney
//
//   task:      This function is given a value in cents.  It will convert cents to dollars
//              and cents which is stored in a local variable called total which is sent
//              back to the calling function through the parameter dollars.  It will keep
```

```
//                  a running total of all the money processed in a local static variable called
//                  sum.
//
//   data in:       cents which is an integer
//   data out:      dollars (which alters the corresponding actual parameter
//
//*********************************************************************


void normalizeMoney(float& dollars,  int cents )
{

        float total=0;

        // Fill in the definition of sum as a static local variable

        _____ sum = 0.0;

        // Fill in the code to convert cents to dollars

        total=total+dollars;
        sum = sum + dollars;

        cout << "We have added another $" << dollar << " to our total" <<endl;
        cout << "Our total so far is $" << sum << endl;

        cout << "The value of our local variable total is $" <<total << endl;
}
```

Exercise 1: You will notice that the function has to be completed. This function will take `cents` and convert it to `dollars`. It also keeps a running total of all the money it has processed. Assuming that the function is complete, write out what you expect the program will print.

Exercise 2: Complete the function. Fill in the blank space to declare `sum` and then write the code to convert `cents` to `dollars`. Example: 789 cents would convert to 7.89. Compile and run the program to get the expected results. Think about how `sum` should be defined.

Problem 4.3 Bring in `billfile.cpp` from the Lab 4 folder. The code is as follows

```
// This program will read in the quantity of a particular item and its price
// It will then print out the total price.
// The input will come from a data file and the output will go to an output file.
```

```
// PLACE YOUR NAME HERE
#include <fstream>
#include <iomanip>
using namespace std;

int main()
{
        ifstream dataIn;                // defines an input stream for a data file
        ofstream dataOut;               // defines an output stream for an output file
        int quantity;                   // contains the amount of items purchased
        float itemprice;                // contains the price of each item
        float totalbill;                // contains the total bill, i.e. The price of all items

        dataIn.open("transaction.dat"); // This opens the file
        dataOut.open("bill.out");

        // Fill in the appropriate code in the blank below
        _________ <<  setprecision(2)  << fixed << showpoint;          //formatted output


        // Fill in the input statement that brings in the quantity and price of the item.

        // Fill in the assignment statement that determines the total bill.

        // Fill in the output statement that prints the total bill, with a label, to an output file

        return 0;
}
```

Exercise 1: This program gets the information from a file and places the output to a file. You must create the data file. Create a data file called `transaction.dat` that has the following information:

```
22
10.98
```

Exercise 2: Fill in the blank and the statements that will read the data from the file and print the following to `bill.out`:

The total bill is $241.56

Problem 4.4 Retrieve program `convertmoney.cpp` from the Lab 4 folder. The code is as follows

```
#include <iostream>
#include <iomanip>
using namespace std;
```

```
// This program will input American money and convert it to foreign currency

// PLACE YOUR NAME HERE

//Prototypes of the functions
void convertMulti(float dollars, float& euros, float& pesos);
void convertMulti(float dollars, float& euros, float& pesos, float& yen);
float convertToYen(float dollars);
float convertToEuros(float dollars);
float convertToPesos(float dollars);

int main ()
{
        float dollars;
        float euros;
        float pesos;
        float yen;

        cout << fixed << showpoint << setprecision(2);


        cout << "Please input the amount of American Dollars you want converted " << endl;
        cout << “to euros and pesos” << endl;
        cin >> dollars;

        // Fill in the code to call convertMulti with parameters dollars, euros, and pesos
        // Fill in the code to output the value of those dollars converted to both euros and pesos

        cout << "Please input the amount of American Dollars you want converted\n";
        cout << " to euros, pesos and yen" << endl;
        cin >> dollars;

        // Fill in the code to call convertMulti with parameters dollars, euros, pesos, and yen
        // Fill in the code to output the value of those dollars converted to yen

        cout << "Please input the amount of American Dollars you want converted\n"
        cout << " to yen" << endl;
        cin >> dollars;

        // Fill in the code to call convertToYen
        // Fill in the code to output the value of those dollars converted to both euros and pesos


        cout << "Please input the amount of American Dollars you want converted\n        "
        cout << " to euros" << endl;
```

```
        cin >> dollars;


        // Fill in the code to call convertToEuros
        // Fill in the code to output the value of those dollars converted to euros

        cout << "Please input the amount of American Dollars you want converted\n"
        cout << " to pesos " << endl;
        cin >> dollars;

        // Fill in the code to call convertToPesos
        // Fill in the code to output the value of those dollars converted to pesos

        return 0;

}

//   All of the functions are stubs that just serve to test the functions
//   Replace with code that will cause the functions to execute properly


//  ***************************************************************
//                              convertMulti
//
//   task:       This function takes a dollar value and converts it to euros and pesos
//   data in:    dollars
//   data out:  euros and pesos
//
//***************************************************************


void convertMulti(float dollars, float& euros, float& pesos)
{

        cout << "The function convertMulti with dollars, euros and pesos "
             << endl <<" was called with " << dollars <<" dollars" << endl << endl;
}




//  ***************************************************************
//                              convertMulti
//
//   task:       This function takes a dollar value and converts it to euros pesos and yen
```

```
//   data in:    dollars
//   data out:  euros pesos yen
//
//*****************************************************************


void convertMulti(float dollars, float& euros, float& pesos, float& yen)
{

        cout << "The function convertMulti with dollars, euros, pesos and yen "
             << endl <<" was called with " << dollars <<" dollars" << endl << endl;
}

//  ***************************************************************
//                              convertToYen
//
//   task:              This function takes a dollar value and converts it to yen
//   data in:           dollars
//   data returned:   yen
//
//*****************************************************************


float convertToYen(float dollars)
{

        cout << "The function convertToYen was called with " << dollars <<" dollars"
             << endl << endl;
        return 0;
}

//  ***************************************************************
//                              convertToEuros
//
//   task:              This function takes a dollar value and converts it to euros
//   data in:           dollars
//   data returned:  euros
//
//*****************************************************************


float convertToEuros(float dollars)
{

        cout << "The function convertToEuros was called with " << dollars <<" dollars"
             << endl << endl;
```

```
        return 0;
}

//  ****************************************************************
//                              convertToPesos
//
//   task:              This function takes a dollar value and converts it to pesos
//   data in:           dollars
//   data returned:  pesos
//
//****************************************************************


float convertToPesos(float dollars)
{

        cout << "The function convertToPesos was called with " << dollars <<" dollars"
             << endl << endl;
        return 0;
}
```

Exercise 1: Run this program and observe the results. You can input anything that you like for the dollars to be converted. Notice that it has stubs as well as overloaded functions. Study the stubs carefully. Notice that in this case the value returning functions always return 0.

Exercise 2: Complete the program by turning all the stubs into workable functions. Be sure to call true functions differently than methods. Make sure that functions return the converted dollars into the proper currency. Although the exchange rates vary from day to day, use the following conversion chart for the program. These values should be defined as constants in the global section so that any change in the exchange rate can be made there and nowhere else in the program.

One Dollar = .701 euros
12.66 pesos
91.25 yen

Sample Run:
Please input the amount of American Dollars you want converted to euros and pesos
9.35
$9.35 is converted to 6.55 euros and 118.37 pesos.

Please input the amount of American Dollars you want converted to euros, pesos and yen
10.67
$10.67 is converted to 7.48 euros and 135.08 pesos and 973.64 yen

Please input the amount of American Dollars you want converted to yen
12.78
$12.78 is converted to 1166.17 yen

Please input the amount of American Dollars you want converted to euros
2.45
$2.45 is converted to 1.72 euros

Please input the amount of American Dollars you want converted to pesos
8.75
$8.75 is converted to 110.78 pesos
Press any key to continue . . .

Student Generated Code Assignments
Student Generated Code Problem Option 4.5

Write a program that outputs a dentist bill. For members of a dental plan, the bill consists of the service charge (for the particular procedure performed), and test fees, input to the program by the user. To nonmembers the charges consist of the above services plus medicine (also input by the user). The program first asks if the patient is a member of the dental plan. The program uses two overloaded functions to calculate the total bill. Both are value returning functions that return the total charge.

Sample Run 1:
Please input a one if you are a member of the dental plan
Input any other number if you are not
1
Please input the service charge
7.89
Please input the test charges
89.56
The total bill is $97.45

Sample Run 2:
Please input a one if you are a member of the dental plan
Input any other number if you are not
2
Please input the service charge
75.84
Please input the test charges
49.78
Please input the medicine charges
40.22
The total bill is $165.84

Lesson 4 Summary

Scope refers to the time during a program that an identifier has storage assigned to it.

Scope Rules

1. The scope of a global identifier, any identifier declared or defined outside all functions, is the entire program.

2. Functions are defined globally. That means any function can call any other function at any time.

3. The scope of a local identifier is from the point of its definition to the end of the block in which it is defined. This includes any nested blocks that may be contained within, unless the nested block has a variable defined in it that has the same name.

4. Scope of formal parameters is the same as the scope of local variables of the function.

Static variables are local variables that retain their value even when the function in which they are defined are not executing. They are still only accessible when the function is in use. Static variables are defined by placing the word **`static`** before the data type and name of the variable as shown below.

```
static int totalPay = 0;
```

Overloading functions

Two or more functions may have the same name as long as their parameters differ in quantity or data type. For example, a program could have two functions with the same name as long as the number of parameters differed or the data types of the parameters were not the same.

A **stub** is nothing more than a dummy function that is called instead of the actual function. It usually does little more than write a message to the screen indicating that it was called with certain arguments.

A **driver** is a module that tests a function by simply calling it with test data.

Files

Input can come from and output could go to files. To do either of these things add the `#include <fstream>` directive in the header to allow files to be created and accessed. A file containing data to be input to the computer should be defined as an `ifstream` data type and an output file should be defined as `ofstream.`

Suppose we have a data file called `grades.dat` that contains three grades and we want to take those grades and output them to an output file that we will call `finalgrade.out`. The following code shows how this can be done in C++:

```
#include <fstream> // This statement is needed to use files
using namespace std;


int main()
{
  float grade1, grade2, grade3; // defines three float variables

  ifstream dataFile;                    // This defines an input file stream.
                                        // dataFile is the "internal" name is used in the
                                        //  program for accessing the data file

  ofstream outFile;                     // This defines an output file stream.
                                        //  outFile is the"internal" name that is used in the
                                        //  program for accessing the output file called

  outFile << fixed << showpoint;      // These can be used with output files as well as cout

  dataFile.open("grades.dat");          // This ties the internal name, dataFile,
                                        // to the actual file, grades.dat

  outFile.open("finalgrade.out")       //  This ties the internal name, outFile, to
                                       //   the actual file, finalgrade.out.

  dataFile >> grade1 >> grade2         //  The reads the values from the input file
           >> grade3;                  //   into the three variables

  outFile << grade1 << endl;           //   These 3 lines write the values stored in the
  outFile << grade2 << endl;           //    3 variables to the output file
  outFile << grade3 << endl;

  datafile.close();                   //  files should be closed before the program ends.
  outfile.close();
```

Lesson 5: Arrays

Arrays

One-Dimensional Arrays

So far we have talked about a variable as a single location in the computer's memory. It is possible to have a collection of memory locations, all of which have the same data type, grouped together under one name. Such a collection is called an **array**. Like every variable, an array must be declared so that the computer can "reserve" the appropriate amount of memory. This amount is based upon the type of data to be stored and the number of locations, i.e., size of the array, each of which is given in the declaration.

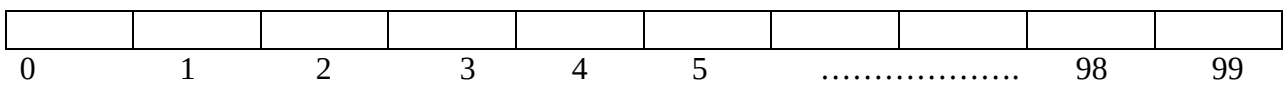
Example: Given a list of ages (from a file or input from the keyboard), find the number of people for each age.

The programmer would not know the ages to be read but would need a space for the total number of occurrences of each "legitimate age." Assuming that ages 1, 2, . . . , 100 are possible, the following array declaration could be used.

```
const int TOTALYEARS = 100;

int main()

{
        int ageFrequency[TOTALYEARS];      //reserves memory for 100 integers
        ….
        return 0;
}
```

Following the rules of variable definition, the data type (integer in this case) is given first, followed by the name of the array (ageFrequency), and then the total number of memory locations enclosed in brackets. The number of memory locations must be an integer expression greater than zero and can be given either as a named constant (as shown in the above example) or as a literal constant (an actual number such as 100).

Each element of an array, consisting of a particular memory location within the group, is accessed by giving the name of the array and a position with the array (subscript). In C++ the subscript, sometimes referred to as index, is enclosed in square brackets. The numbering of the subscripts always begins at 0 and ends with one less than the total number of locations. Thus the elements in the `ageFrequency` array defined above are referenced as `ageFrequency[0]` through `ageFrequency[99]`.



If in our example we wanted ages from 1 to 100, the number of occurrences of age 4 would actually

be placed in subscript 3 since it is the "fourth" location in the array. This odd way of numbering is often confusing to new programmers; however, it quickly becomes routine. Some students actually add one more location and then ignore location 0, letting 1 be the first location. In the above example such a process would use the following declaration:
`int agefrequency[101];` and use only the subscripts 1 through 100. Our examples will use location 0.

Array Initialization

In our example, `ageFrequency[0]` keeps a count of how many 1's we read in, `ageFrequencey[1]` keeps count of how many 2's we read in, etc. Thus, keeping track of how many people of a particular age exist in the data read in would require reading each age and then adding one to the location holding the count for that age. Of course it is important that all the counters start at 0. The following shows the initialization of all the elements of our sample array to 0.

```
for (int pos = 0; pos < TOTALYEARS; pos++)   // pos acts as the array subscript
        ageFrequency[pos] = 0;     // only one statement so { } not needed
```

A simple for loop will process the entire array, adding one to the subscript each time through the loop. Notice that the subscript (pos) starts with 0. Why is the condition pos < TOTALYEARS used instead of pos <= TOTALYEARS? Remember that the last subscript is one less than the total number of elements in the array. Hence the subscripts of this array go from0 to 99.

Array Processing

Arrays are generally processed inside loops so that the input/output processing of each element of the array can be performed with minimal statements. Our age frequency problem first needs to read in the ages from a file or from the keyboard. For each age read in, the "appropriate" element of the array (the one corresponding to that age) needs to be incremented by one. The following examples show how this can be accomplished:

from a file using infile **as a logical name**

```
infile >> currentage;
while (infile)

{
    agefrequency[currentage-1] =
    agefrequency[currentage-1] + 1;
    infile >> currentage;


}
```

from a keyboard with –99 as sentinel data

```
cout << "Please input an age from one"
       << " to 100.  Input -99 to stop" << endl;
 cin >> currentage;

while (currentage != -99)
{
    agefrequency[currentage-1] =
    agefrequency[currentage-1] + 1;
    cout << "Please input an age from one "
          <<" to 100. Put -99 to stop" << endl;
     cin >> currentage;
}
```

The `while(infile)` statement means that while there is more data in the file `infile`, the loop will continue to process.

To read from a file or from the keyboard we prime the read, which means the first value is read in before the test condition is checked to see if the loop should be executed. When we read an age, we increment the location in the array that keeps track of the amount of people in the corresponding age group. Since C++ array indices always start with 0, that location will be at the subscript one value less than the age we read in.

4	0	14	5	0	6			1	0
0	1	2	3	4	5 ……			98	99
1 year	2 years	3 years	4 years	5 years	6 years			99 years	100 years

Each element of the array contains the number of people of the given age. The data shown here is from a random sample run. In writing the information stored in the array, we want to make sure that only those array elements that have values greater than 0 are output. The following code will do this.

```
for (int ageCounter = 0; ageCounter < TOTALYEARS; ageCounter++)
      if (ageFrequency[ageCounter] > 0)
         cout << "The number of people " << ageCounter + 1 <<" years old is "
              << ageFrequency[ageCounter] << endl;
```

The `for` loop goes from 0 to one less than `TOTALYEARS` (0 to 99). This will test every element of the array. If a given element has a value greater than 0, it will be output. What does outputting `ageCounter + 1` do? It gives the age we are dealing with at any given time, while the value of `ageFrequency[ageCounter]` gives the number of people in that age group.

Arrays as Arguments

Arrays can be passed as arguments (parameters) to functions. Although variables can be passed by value or reference, arrays are always **passed by pointer**, which is similar to pass by reference, since it is not efficient to make a "copy" of all elements of the array. This means that array, like pass by reference parameters, can be altered by the called function. However, they NEVER have the & symbol between the data type and name, as pass by reference parameters do.
If we don't want the called function to change elements of an array we use the word const before the data type on the formal parameter list.

Example 5.1 illustrates how arrays are passed as arguments to functions.

```
/*************************************************************************
                    The grade average program
This program illustrates how one-dimensional arrays are used and how they are passed as arguments to
functions.  It contains two functions.  The first function is called to allow the user to input a set of
grades and store them in an array.  The second function is called to find the average grade

*************************************************************************/

#include <iostream>
using namespace std;
```

```
const int TOTALGRADES=50;   // TOTALGRADES is the maximum size of the array

// function prototypes

void getData(int array[], int& sizeOfArray);  // This procedure will read values into the array

float findAverage(const int array[], int sizeOfArray);
  //  This procedure will find the average of values stored in an array.  Notice that the word
  //   const in front of the data type of the array prevents the function from altering the array

int main()
{
        int grades[TOTALGRADES];    // defines an array that holds up to 50 integers
        int numberOfGrades = 0;          //  holds the value of the number of grades read in
        float average;                   // holds the average of all grades read in

        getData(grades, numberOfGrades);   // getData is called to read the grades into the array
                                           // and to store the number of grades read in
        average = findAverage(grades, numberOfGrades);

        cout << endl << "The average of the " << numberOfGrades  << " grades read in is "
             <<  average <<  "."  << endl << endl;

        return 0;
}

//*************************************************************************
//                                    getData
//
// task:          This function inputs and stores data in the grades array.
// data in:       none (the parameters contain no information needed by the getData function
// data out:      an array containing grades and the number of grades
//*************************************************************************

void getData(int array[], int& sizeOfArray)
{
        int pos = 0;                    // array index which starts at 0
        int grade;                      // holds each individual grade read in

        cout << "Please input a grade or type -99 to stop:  " << endl;
        cin >> grade;

        while (grade != -99)
        {
                array[pos] = grade;     // store grade read in to next array location
                pos++;                  // increment array index
```

```
            cout << "Please input a grade or type -99 to stop:  " << endl;
            cin >> grade;
        }

        sizeOfArray = pos:          // upon exiting the loop, pos hold the number of
                                    // grades read in, which is sent to the calling function.
}

//*************************************************************************
//                                      findAverage
//
//  task:               This function finds and returns the average of the values
//
//  data in:            the array containing grades and the array size
// data returned:       the average of the grades contained in that array
//
//*************************************************************************


float findAverage (const int array[], int sizeOfArray)
{
        int sum = 0;                        // holds the sum of all grades in the array

        for (int pos = 0;  pos < sizeOfArray;   pos++)
        {
            sum = sum + array[pos];    // add grade in array position pos to sum
        }

        return  float(sum) / sizeOfArray;
}
```

Notice that a set of empty brackets [] follows the formal parameter of an array which indicates that the data type of this parameter is in fact an array. Notice also that no brackets appear in the call to the functions.

Since arrays in C++ are passed by pointer, which is similar to pass by reference, it allows the original array to be altered, even though no `&` is used to designate this. The `getData` function is thus able to store new values into the array. There may be times when we do not want the function to alter the values of the array. Inserting the word **const** before the data type on the formal parameter list prevents the function from altering the array even though it is passed by pointer. This is why in the preceding example, the `findAverage` function and header had the word const in front of the data type of the array.

```
float findAverage (const int array[], int sizeOfArray);    // prototype
float findAverage (const int array[], int sizeOfArray)      // function header
```

The variable `numberOfGrades` contains the number of elements in the array to be processed. In most cases not every element of the array is used, which means the size of the array given in its definition and the number of actual elements used are rarely the same. For that reason we often pass the actual number of elements used in the array as a parameter to a procedure that uses the array. The variable `numberOfGrades` is explicitly passed by reference (by using &) to the `getData` function where its corresponding formal parameter is called `sizeOfArray`.

Prototypes can be written without named parameters. Function headers must include named parameters.

float findAverage (const int [], int); // prototype without named parameters

The use of brackets in function prototypes and headings can be avoided by declaring a programmer defined datatype. This is done in the global section with a **typedef** statement.

Example: `typedef int GradeType[50];`

This declares a data type, called `GradeType`, that is an array containing 50 integer memory locations. Since `GradeType` is a data type, it can be used in defining variables. The following defines `grades` as an integer array with 50 elements.

GradeType grades;

It has become a standard practice (although not a requirement) to use an uppercase letter to begin the name of a data type. It is also helpful to include the word "type" in the name to indicate that it is a data type and not a variable.

Example 5.2 shows the revised code (in bold) of Example 5.2 using `typedef`.

```
/*************************************************************************
                        The grade average program
This program illustrates how one-dimensional arrays are used and how they are passed as arguments to
functions.  It contains two functions.  The first function is called to allow the user to input a set of
grades and store them in an array.  The second function is called to find the average grade

*************************************************************************/

#include <iostream>
using namespace std;

const int TOTALGRADES=50;   // TOTALGRADES is the maximum size of the array

// function prototypes
```

```
typedef int GradeType[TOTALGRADES];   // declaration of an integer array data type
                                      // called GradeType

void getData(GradeType array, int& sizeOfArray);  // This procedure will read values into the array

float findAverage(const GradeType array,  int sizeOfArray);
  //  This procedure will find the average of values stored in an array.  Notice that the word
  //   const in front of the data type of the array prevents the function from altering the array

int main()
{
        GradeType grades;               // defines an array that holds up to 50 integers
        int numberOfGrades = 0;         //  holds the value of the number of grades read in
        float average;                  // holds the average of all grades read in

        getData(grades, numberOfGrades);   // getData is called to read the grades into the array
                                           // and to store the number of grades read in
        average = findAverage(grades, numberOfGrades);

        cout << endl << "The average of the " << numberOfGrades  << " grades read in is "
             <<  average <<  "."  << endl << endl;

        return 0;
}

//*************************************************************************
//                                  getData
//
// task:          This function inputs and stores data in the grades array.
// data in:       none (the parameters contain no information needed by the getData function
// data out:      an array containing grades and the number of grades
//*************************************************************************

void getData(GradeType array,  int& sizeOfArray)
{
        int pos = 0;                    // array index which starts at 0
        int grade;                      // holds each individual grade read in

        cout << "Please input a grade or type -99 to stop:  " << endl;
        cin >> grade;

        while (grade != -99)
        {
                array[pos] = grade;     // store grade read in to next array location
                pos++;                  // increment array index
```

```
            cout << "Please input a grade or type -99 to stop:  " << endl;
            cin >> grade;
        }

        sizeOfArray = pos:          // upon exiting the loop, pos hold the number of
                                    // grades read in, which is sent to the calling function.
}

//*************************************************************************
//                                      findAverage
//
//  task:               This function finds and returns the average of the values
//
//  data in:            the array containing grades and the array size
// data returned:       the average of the grades contained in that array
//
//*************************************************************************


float findAverage (const GradeType array,  int sizeOfArray)
{
        int sum = 0;                        // holds the sum of all grades in the array

        for (int pos = 0;  pos < sizeOfArray;   pos++)
        {
            sum = sum + array[pos];    // add grade in array position pos to sum
        }

        return  float(sum) / sizeOfArray;
}
```

This method of using **`typedef`** to eliminate brackets in function prototypes and headings is especially useful for multi-dimensional arrays such as those now introduced.

Two-Dimensional Arrays

Data is often contained in a table of rows and columns that can be implemented with a two-dimensional array. Suppose we wanted to read data representing profits (in thousands) for a particular year and quarter.

Quarter 1	Quarter 2	Quarter 3	Quarter 4
72	80	10	100
82	90	43	42
10	87	48	53

This can be done using a **two-dimensional array**.

Example: 5.3

```
const NO_OF_ROWS = 3;
const NO_OF_COLS = 4;

typedef float ProfitType[NO_OF_ROWS][NO_OF_COLS];
   //   declares a new data type which is a 2 dimensional array of floats

int main()
{
        ProfitType profit; // decfines profit as a 2 dimensional array

        for (int row_pos = 0; row_pos < NO_OF_ROWS; row_pos++)
            for (int col_pos = 0; col_pos < NO_OF_COLS; col_pos++)
            {
                cout << "Please input a profit" << endl;
                cin >> profit[row_pos][col_pos];
            }
        return 0;
}
```

A two dimensional array normally uses two loops (one nested inside the other) to read, process or output data.

How many times will the code above ask for a profit? It processes the inner loop `NO_OF_ROWS * NO_OF_COLS` times, which is 12 times in this case.

Multi-Dimensional Arrays

C++ arrays can have any number of dimensions (although more than three is rarely used). To input, process or output every item in an *n*-dimensional array, you need *n* nested loops.

Arrays of Strings

Any variable declared as **char** holds only one character. To hold more than one character in a single variable, that variable needs to be an array of characters. A string (a group of characters that usually form meaningful names or words) is really just an array of characters.

Lab 5: Arrays

The purpose of this lab is to give the student practical applications in working with single and multi-dimensional arrays and to introduce and work with the `typedef` statement.

Create a Lab 5 folder.

Problem 5.1

Retrieve program `testscore.cpp` from the Lab 5 folder. The code is as follows:

```
// This program will read in a group of test scores( positive integers from 1 to 100)
// from the keyboard and then calculates and outputs  the average score
// as well as the highest and lowest score. There will be a maximum of 100 scores.

// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

typedef int GradeType[100];  // declares a new data type:
                             // an integer array of 100 elements


float findAverage (const GradeType, int);  // finds average of all grades
int   findHighest (const GradeType, int);  // finds highest of all grades
int   findLowest  (const GradeType, int);  // finds lowest of all grades

int main()

{
        GradeType  grades;                  // the array holding the grades.
        int  numberOfGrades;                // the number of grades read.
        int pos;                            // index to the array.

        float avgOfGrades;                  // contains the average of the grades.
        int highestGrade;                   // contains the highest grade.
        int lowestGrade;                    // contains the lowest grade.

        // Read in the values into the array

        pos = 0;
        cout << "Please input a grade from 1 to 100, (or -99 to stop)" << endl;
        cin  >> grades[pos];
```

```
        while (grades[pos] != -99)
        {

            // Fill in the code to read the grades

        }

        numberOfGrades = _________;  // Fill blank with appropriate identifier

        // call to the function to find average

        avgOfGrades = findAverage(grades, numberOfGrades);
        cout << endl << "The average of all the grades is " << avgOfGrades << endl;

        //  Fill in the call to the function that calculates highest grade

        cout << endl << "The highest grade is " << highestGrade << endl;


        // Fill in the call to the function that calculates lowest grade
       // Fill in code to write the lowest to the screen

        return 0;
}

//*********************************************************************************
//                          findAverage
//
// task:         This function receives an array of integers and its size.
//              It finds and returns the average of the numbers in the array
// data in:      array of floating point numbers
// data returned: avarage of the numbers in the array
//
//*********************************************************************************

float findAverage (const GradeType  array, int size)

{

        float sum = 0;   // holds the sum of all the numbers

   for (int pos = 0; pos < size; pos++)
           sum = sum + array[pos];
   return (sum / size);  //returns the average

}
```

```
//*************************************************************************
//                          findHighest
//
// task:         This function receives an array of integers and its size.
//               It finds and returns the highest value of the numbers in
//               the array
// data in:      array of floating point numbers
// data returned: highest value of the numbers in the array
//
//*************************************************************************

int   findHighest (const GradeType array, int size)

{

  // Fill in the code for this function

}


//*************************************************************************
//                          findLowest
//
// task:                    This function receives an array of integers and its size.
//                          It finds and returns the lowest value of the numbers in
//                          the array
// data in:                 array of floating point numbers
// data returned: lowest value of the numbers in the array
//
//*************************************************************************

int   findLowest  (const GradeType array, int size)

{
  // Fill in the code for this function

}
```

Exercise 1: Complete this program by filling in the code (places in bold)
Exercise 2: Run the program with the following data: 90 45 73 21 62 –99.
Record the output here:

Exercise 3: Modify your program from Exercise 1 so that it reads the information from the gradfile.txt file, reading until the end of file is encountered. You will need to retrieve this file from the Lab 5 folder and place it in the same folder as your C++ source code. The output should be the same as the output from Exercise 2.

Problem 5.2

Retrieve program `student.cpp` from the Lab 5 folder. The code is as follows

```
// This program will input an undetermined number of student names
// and a number of grades for each student. The number of grades is
// given by the user. The grades are stored in an array.
// Two functions are called for each student.
// One function will give the numeric average of their grades.
// The other function will give a letter grade to that average.
// Grades are assigned on a 10 point spread.
// 90-100 A   80- 89 B  70-79 C   60-69 D   Below 60 F

// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

const  int MAXGRADE = 25;                    // maximum number of grades per student
const  int MAXCHAR = 30;                     // maximum characters used in a name

typedef char StringType30[MAXCHAR + 1];      // character array data type used for names
                                             // having 30 characters or less.
typedef  float GradeType[MAXGRADE];          // one dimensional integer array data type

float findGradeAvg(GradeType, int);          // finds grade average by taking array of
                                             // grades and number of grades as parameters

char  findLetterGrade(float);          // finds letter grade from average given  to it as a parameter

int main()
{
   StringType30 firstname, lastname;      // two arrays of characters defined
   int numOfGrades;                       // holds the number of grades
  GradeType  grades;                      // grades is defined as a one dimensional array
  float average;                          // holds the average of a student's grade
  char moreinput;                         // determines if there is more input

   // Input the number of grades for each student

   cout << "Please input the number of grades each student will receive." << endl
        << "This number must be a number between 1 and " << MAXGRADE << " inclusive" << endl;

    cin >> numOfGrades;
```

```
        while (numOfGrades > MAXGRADE || numOfGrades < 1)
        {
                cout << "Please input the number of grades for each student." << endl
                    << "This number must be a number between 1 and " << MAXGRADE
                     << " inclusive" << endl;

                cin >> numOfGrades;

        }

    // Input names and grades for each student

        cout << "Please input a y if you want to input more students"
                  << " any other character will stop the input" << endl;
        cin >> moreinput;

        while ( moreinput == 'y' || moreinput == 'Y')

        {
                cout << "Please input the first name of the student" << endl;
                cin >> firstname;
                cout << endl << "Please input the last name of the student" << endl;
                cin >> lastname;

                for (int count = 0; count < numOfGrades; count++)

                {

                        cout << endl << "Please input a grade" << endl;

                        // Fill in the input statement to place grade in the array

                }

                 cout << firstname << ' ' << lastname << " has an average of ";

                // Fill in code to get and print average of student to screen
                // Fill in call to get and print letter grade of student to screen

                cout << endl << endl << endl;
                cout << "Please input a y if you want to input more students"
                       << " any other character will stop the input" << endl;
                 cin >> moreinput;

        }

        return 0;
}
```

```
//*************************************************************************
//                        findGradeAvg
//
//  task:               This function finds the average of the
//                      numbers stored in an array.
//
//  data in:            an array of integer numbers
//  data returned:   the average of all numbers in the array
//*************************************************************************

float findGradeAvg(GradeType array, int numgrades)

{
   // Fill in the code for this function
}

//*************************************************************************
//                        findLetterGrade
//
//  task:                    This function finds the letter grade for the number
//                            passed to it by the calling function
//
//  data in:                  a floating point number
//  data returned:           the grade (based on a 10 point spread) of the number
//                           passed to the function
//
//*************************************************************************

char  findLetterGrade(float numgrade)
{
  // Fill in the code for this function
}
```

Exercise 1: Complete the program by filling in the code. (Areas in bold)
Run the program with 3 grades per student using the sample data below.

Mary Brown 100 90 90
George Smith 90 30 50
Dale Barnes 80 78 82
Sally Dolittle 70 65 80
Conrad Bailer 60 58 71

You should get the following results:

Mary Brown has an average of 93.333 which gives the letter grade of A
George Smith has an average of 56.6667 which gives the letter grade of F
Dale Barnes has an average of 80 which gives the letter grade of B
Sally Dolittle has an average of 71.6667 which gives the letter grade of C
Conrad Bailer has an average of 63 which gives the letter grade of D

Problem 5.3

Look at the following table containing prices of certain items:

12.78	23.78	45.67	12.67
7.83	4.89	5.99	56.84
13.67	34.84	16.71	50.89

These numbers can be read into a two-dimensional array.
Retrieve price.cpp from the Lab 5 folder. The code is as follows:

```
// This program will read in prices and store them into a two-dimensional array
// It will print those prices in a table form.

// PLACE YOUR NAME HERE

#include <iostream>
#include <iomanip>
using namespace std;

const  MAXROWS = 10;
const  MAXCOLS = 10;

typedef float PriceType[MAXROWS][MAXCOLS];    // creates a new data type
                                              // of a 2D array of floats

void   getPrices(PriceType, int&, int&);        // gets the prices into the array
void   printPrices(PriceType, int, int);        // prints data as a table

int main()

{
        int rowsUsed;                    // holds the number of rows used
        int colsUsed;                   // holds the number of columns used
        PriceType priceTable;           // a 2D array holding the prices

        getPrices(priceTable, rowsUsed, colsUsed);  // calls getPrices to fill the array
        printPrices(priceTable, rowsUsed, colsUsed);// calls printPrices to display array

        return 0;
}
```

```
//*********************************************************************************
//                      getPrices
//
//  task:     This procedure asks the user to input the number of rows and
//            columns. It then asks the user to input (rows * columns) number
//            of prices where x = rows * columns.  The data is placed in the array.
//  data in:  none
//  data out: an array filled with numbers and the number of rows
//            and columns used.
//
//*********************************************************************************

void   getPrices(PriceType table, int& numOfRows, int& numOfCols)
{
          cout << "Please input the number of rows from 1 to "<< MAXROWS << endl;
          cin >> numOfRows;

          cout << "Please input the number of columns from 1 to "<< MAXCOLS << endl;
          cin >> numOfCols;


          for (int row = 0;  row < numOfRows;  row++)
          {
                    for (int col = 0; col < numOfCols;  col++)

                              // Fill in the code to read and store the next value in the array
          }
}



//*********************************************************************************
//                      printPrices
//
//  task:     This procedure prints the table of prices
//  data in:  an array of floating point numbers and the number of rows
//                              and columns used.
//  data out: none
//
//*********************************************************************************

void   printPrices(PriceType table, int numOfRows, int numOfCols)


{
```

```
        cout << fixed << showpoint << setprecision(2);

        for (int row = 0;  row < numOfRows; row++)
        {
                for (int col = 0;  col < numOfCols; col++)

                        // Fill in the code to print the table
        }
}
```

Exercise 1: Fill in the code to complete both functions `getPrices` and `printPrices` then run the program with the following data:

Please input the number of rows from 1 to 10
2

Please input the number of columns from 1 to 10
3

Please input the price of an item with 2 decimal places
1.45

Please input the price of an item with 2 decimal places
2.56

Please input the price of an item with 2 decimal places
12.98

Please input the price of an item with 2 decimal places
37.86

Please input the price of an item with 2 decimal places
102.34

Please input the price of an item with 2 decimal places
67.89

1.45	**2.56**	**12.98**
37.86	**102.34**	**67.89**

Exercise 2: Why does `getPrices` have the parameters `numOfRows` and `numOfCols` passed by reference whereas `printPrices` has those parameters passed by value?

Exercise 3: The following code is a function that returns the highest price in the array. After studying it very carefully, place the function in the above

program and have the program print out the highest value.

```
float HighestPrice(PriceType table, int numOfRows, int numOfCols)

// This function returns the highest price in the array
{
        float highestPrice;
        highestPrice = table[0][0]; // make first element the highest price
        for (int row = 0; row < numOfRows; row++)
                for (int col = 0; col < numOfCols; col++)
                        if ( highestPrice < table[row][col] )
                                highestPrice = table[row][col];
        return highestPrice;
}
```

NOTE: This is a value returning function. Be sure to include its prototype in the global section.

Exercise 4: Create another value returning function that finds the lowest price in the array and have the program print that value.

Exercise 5: After completing all the exercises above, run the program again with the values from Exercise 1 and record your results.

Problem 5.4

Look at the following table that contains quarterly sales transactions for three years of a small company. Each of the quarterly transactions are integers (number of sales) and the year is also an integer.

YEAR	Quarter1	Quarter2	Quarter3	Quarter4
2000	72	80	60	100
2001	82	90	43	98
2002	64	78	58	84

We could use a two-dimensional array consisting of 3 rows and 5 columns.
Even though there are only four quarters we need 5 columns (the first column holds the year).

Retrieve quartsal.cpp from the Lab 5 folder. The code is as follows:

```
// This program will read in the quarterly sales transactions for a given number
// of years. It will print the year and transactions in a table format.
// It will calculate year and quarter total transactions.


// PLACE YOUR NAME HERE
```

```
#include <iostream>
#include <iomanip>
using namespace std;

const  MAXYEAR = 10;
const  MAXCOL = 5;

typedef int SalesType[MAXYEAR][MAXCOL];   // creates a new 2D integer data type

void   getSales(SalesType, int&);              // places sales figures into the array
void   printSales(SalesType, int);             // prints data as a table
void   printTableHeading();                    // prints table heading

int main()
{
        int yearsUsed;                  // holds the number of years used
        SalesType sales;                // 2D array holding
                                        // the sales transactions
        getSales(sales, yearsUsed);     // calls getSales to put data in array
        printTableHeading();            // calls procedure to print the heading
        printSales(sales, yearsUsed);   // calls printSales to display table

        return 0;
}

//*************************************************************************
//                      printTableHeading
//
//  task:     This procedure prints the table heading
//  data in:  none
//  data out: none
//
//*************************************************************************

void printTableHeading()
{
        cout << setw(30) << "YEARLY QUARTERLY SALES" << endl << endl << endl;

        cout << setw(10) << "YEAR" << setw(10) << "Quarter 1"
                << setw(10) << "Quarter 2" << setw(10) << "Quarter 3"
                << setw(10) << "Quarter 4" << endl;
}
```

```
//*************************************************************************
//                      getSales
//
//  task:     This procedure asks the user to input the number of years.
//            For each of those years it asks the user to input the year
//            (e.g. 2004), followed by the sales figures for each of the
//            4 quarters of that year.  That data is placed in a 2D array
//  data in:  a 2D array of integers
//  data out: the total number of years
//
//*************************************************************************

void   getSales(SalesType  table, int&  numOfYears)
{
        cout << "Please input the number of years (1-" << MAXYEAR << ')' << endl;
        cin >> numOfYears;


        // Fill in the code to read and store the next value

}

//*************************************************************************
//                      printSales
//
//  task:     This procedure prints out the information in the array
//  data in:  an array containing sales information
//  data out: none
//
//*************************************************************************

void   printSales(SalesType table, int numOfYears)
{
        // Fill in the code to print the table
}
```

Exercise 1: Fill in the code for both `GetSales` and `PrintSales`.

Exercise 2: Run the program so that the following chart is printed.

YEAR	Quarter1	Quarter2	Quarter3	Quarter4
2000	72	80	60	100
2001	82	90	43	98
2002	64	78	58	84

Student Generated Code Assignments

Student Generated Code Option 5.5

Given a list of ages (1 to 100) from the keyboard, write a complete age population program that will tally how many people are in each age group.

Sample Run:
Please input an age from one to 100, put -99 to stop
5
Please input an age from one to 100, put -99 to stop
10
Please input an age from one to 100, put -99 to stop
100
Please input an age from one to 100, put -99 to stop
20
Please input an age from one to 100, put -99 to stop
5
Please input an age from one to 100, put -99 to stop
8
Please input an age from one to 100, put -99 to stop
20
Please input an age from one to 100, put -99 to stop
5
Please input an age from one to 100, put -99 to stop
9
Please input an age from one to 100, put -99 to stop
17
Please input an age from one to 100, put -99 to stop
-99

The number of people 5 years old is 3
The number of people 8 years old is 1
The number of people 9 years old is 1
The number of people 10 years old is 1
The number of people 17 years old is 1
The number of people 20 years old is 2
The number of people 100 years old is 1

Student Generated Code Option 5.6
Write a program that will input temperatures for consecutive days.
The program will store these values into an array and call a function that will return the average of the temperatures. It will also call a function that will return the highest temperature and a function that will return the lowest temperature. The user will input the number of temperatures to be read. There will be no more than 50 temperatures. Use `typedef` to declare the array type. The average should be displayed to two decimal places.
Sample Run:

Please input the number of temperatures to be read
5
Input temperature 1:
68
Input temperature 2:
75
Input temperature 3:
36
Input temperature 4:
91
Input temperature 5:
84

The average temperature is 70.80
The highest temperature is 91.00
The lowest temperature is 36.00

Student Generated Code Option 5.7

Write a program that will input letter grades (A, B, C, D, F), the number of which is input by the user (a maximum of 50 grades). The grades will be read into an array. A function will be called five times (once for each letter grade) and will return the total number of grades in that category. The input to the function will include the array, number of elements in the array and the letter category (A,B, C, D or F). The program will print the number of grades that are A, B, etc.
Sample Run:
Please input the number of grades to be read in. (1-50)
6
All grades must be upper case A B C D or F
Input a grade
A
Input a grade
C
Input a grade
A
Input a grade
B
Input a grade
B
Input a grade
D
Number of A=2
Number of B=2
Number of C=1
Number of D=1
Number of F=0

Lesson 5 Summary

Array - An array is a sequence of memory locations of the same type each of which is accessed by the same name, each distinguished by an index.

Array declaration is given by a data type followed by a name followed by a number (in the form of a constant, variable or literal) enclosed in brackets.
const int TOTALYEARS = 100;
int ageFrequencey[TOTALYEARS];

Arrays are initialized and processed in loops.

Initialization can be done as follows:

```
for (int pos=0; pos < TOTALYEARS; pos++)
   ageFrequency[pos] = 0;
```

Arrays can be passed as arguments but they are always passed by pointer which is similar to pass by reference. This means that the called function could alter elements within the array.
To prevent an array from being altered by the called function the word **const** is placed in front of the data type in the prototype and heading of the called function. An array formal parameter is indicated by including brackets [] after the name of the parameter.

typedef is the statement that allows the programmer to declare a new data type. It is often used to create an array data type.

```
float findAverage (const int array[], int sizeOfArray);    prototype
float findAverage (const int array[], int sizeOfArray)     function heading
```

```
typedef int GradeType[50];
```

It is a standard practice to use an uppercase letter to begin the name of a data type and to include the word type in the name. When using the typedef statement array parameters do not have to include the brackets.

float findAverage (GradeType name OfArray, int sizeOfArray)

Arrays can be multi-dimensional

Example of a two dimensional array definition.

const NO_OF_ROWS = 3;
const NO_OF_COLS =4;

typedef float ProfitType {NO_OR_ROWS] [NO_OF_COLS];

Lesson 6: Searching and Sorting Arrays

Search Algorithms

A search algorithm is a procedure for locating a specific datum from a collection of data.

For example, suppose you want to find the phone number for Wilson Electric in the phonebook. You open the phonebook to the business section under W and then look for all the entries that begin with the word Wilson. There are numerous such entries, so you look for the one(s) that end with Electric. This is an example of a **search algorithm**. Since each section in the phonebook is alphabetized, this is a particularly easy search. Of course, there are numerous types of "collections of data" that one could search. In this section we will focus on searching arrays. Two algorithms, the linear and binary searches, will be studied. We will see that each algorithm has its advantages and disadvantages.

Linear Search

The easiest array search to understand is probably the **linear search**. This algorithm starts at the beginning of the array and then steps through the elements sequentially until either the desired value is found or the end of the array is reached. For example, suppose we want to find the first occurrence of the letter "o" in the word "harpoon." We can visualize the corresponding character array as follows:

H	a	r	p	o	o	n	\0
0	1	2	3	4	5	6	7

In C++ we can initialize the character array with the desired string:
char word[8] = "Harpoon";

So `word[0]='H', word[3]= 'p'`, and `word[7] = '\0'`. The `'\0'` marks the end of the string and is called the null character. It is discussed further in a later lesson. If we perform a linear search looking for 'o,' then we would first check `word[0]` which is not equal to `'o'`. So we would then move to `word[1]` which is also not equal to `'o'`. We continue until we get to `word[4]='o'`. At this point the subscript 4 is returned so we know the position in the array that contains the first occurrence of the letter 'o'. What would happen if we searched for 'z'? Certainly we would step through the array until we reached the end and not find any occurrence of 'z'. What should the search function return in this case? It is customary to return –1 since this is not a valid array subscript. Here is the complete program that performs the linear search:

Example 6.1:

```
// This program performs a linear search on a character array

#include <iostream>
using namespace std;
```

```
int searchList( char[], int, char);          // function prototype
const int SIZE = 8;

int main()
{
        char word[SIZE] = "Harpoon";
        int found;
        char ch;

        cout << "Enter a letter to search for:" << endl;
        cin >> ch;

        found = searchList(word, SIZE, ch);
        if (found  ==  -1)
                cout << "The letter " << ch << " was not found in the list" << endl;
        else
                cout << "The letter " << ch << " is in the " << found + 1
                        << " position of the list" << endl;

       return 0;
}

//*********************************************************************
//                              searchList
//
//  task:               This searches an array for a particular value
//  data in:            List of values in an array, the number of elements in the array,
//                      and the value searched for in the array
// data returned:       Position in the array of the value or -1 if value not found
//
//*********************************************************************

int searchList( char list[], int numElems, char value)
{
        for (int count = 0; count <= numElems; count++)    // for loop steps through the array
        {
                if (list[count] == value)      // each array entry is checked to see if  it contains the
                                               // desired value.
                return count;    // if the desired value is found, the array subscript count is returned to
                                 // indicate the location in the array.
          }
        return -1;                // if the value is never found, -1 is  returned.
}
```

For example, suppose we wish to search the word “Harpoon” for the letter 'o'. The function `searchList` does the linear search and returns the index 4 of the array where 'o' is found. However,

the program outputs 5 for the position since we want to output the character's position within the string rather than its storage location in the word array. You have certainly noticed that there is a second occurrence of 'o' in the word "Harpoon." However, the linear search does not find it since it quits after finding the first occurrence.

One advantage of the linear search is its simplicity. It is easy to step sequentially through an array and check each element for a designated value. Another advantage is that the elements of the array do not need to be in any order to implement the algorithm. For example, to search the integer arrays

First Array	23	45	12	456	99
Second Array	12	29	45	23	456

for the integer 99, the linear search will work. It will return 4 for the first array and –1 for the second. The main disadvantage of the linear search is that it is time consuming for large arrays. If the desired piece of data is not in the array, then the search has to check every element of the array before it returns –1. Even if the desired piece of data is in the array, there is a very good chance that a significant portion of the array will need to be checked to find it. So we need a more efficient search algorithm for large arrays.

The Binary Search

A more efficient algorithm for searching an array is the **binary search** which eliminates half of the array every time it does a check. The drawback is that the data in the array must be ordered to use a binary search. If we are searching an array of integers, then the values stored in the array must be arranged in order from largest to smallest or smallest to largest.

Examples: Consider the following three integer arrays:

1)	19	15	13	13	11	6	-1	-3
2)	19	15	16	13	13	11	-1	-3
3)	-3	0	1	1	12	14	18	25

The arrays in 1) and 3) could be searched using a binary search. In 1) the values are arranged largest to smallest and in 3) the values are arranged smallest to largest. However, the array in 2) could not be searched using a binary search due to the first three elements of the array: the values of the elements decrease from 19 to 15 but then increase from 15 to 16.

Now that we know which types of arrays are allowed, let us next describe what the binary search actually does. For the sake of argument, let us assume the values of an integer array are arranged from smallest to largest and the integer we are searching for is stored in the variable called `wanted`. We first pick an element in the middle of the array—let us call it `middle`. Think about how the number, whether it is even or odd, of elements in the array affects this choice. If `middle = wanted`, then we are done. Otherwise, `wanted` must be either greater than or less than `middle`. If `wanted < middle`, then since the array is in ascending order we know that `wanted` must be before `middle` in the array so we can ignore the second half of the array and search the first half. Likewise, if `wanted > middle`, we

can ignore the first half of the array and search just the second half. In both cases we can immediately eliminate half of the array. Once we have done this, we will choose the middle element of the half that is left over and then repeat the same process until either `wanted` is found or it is determined that `wanted` is not in the array.

The following program performs a binary search on an array of integers that is ordered from largest to smallest. Students should think about the logic of this search and how it differs from the argument given above for data ordered smallest to largest.

Example 6.2

```
// This program demonstrates a Binary Search

#include <iostream>
using namespace std;

int binarySearch(int [], int, int);          // function prototype

const int SIZE = 16;

int main()
{
        int found, value;
        int array[] = {34,19,19,18,17,13,12,12,12,11,9,5,3,2,2,0};   //array to be searched

        cout << "Enter an integer to search for: " << endl;
        cin >> value;

        found = binarySearch(array, SIZE, value);
         //function call to perform the binary search on array of SIZE elements looking for an
         //occurrence of value

        if (found == -1)
                cout << "The value " << value << " is not in the list" << endl;
        else
                cout << "The value " << value << " is in position number "
                        << found + 1 << " of the list" << endl;
        return 0;
}
```

```
//*********************************************************************
//                          binary Search
//
//  task:              This function searches an array for a particular value
//  data in:           List of values in an ordered array, the number of elements in the
//                     array, and the value searched for in the array
//  data returned:     Position in the array of the value or -1 if value not found
//
//*********************************************************************

int binarySearch(int array[], int numElems, int value)          // function heading
{
        int first = 0;                                          // First element of list
        int last = numElems - 1;                                // last element of the list
        int middle;                                             // variable containing the current
                                                                // middle value of the list
        while (first <= last)
        {
                middle = first + (last - first) / 2;            // finds the middle element of the array

                if (array[middle] == value)
                        return middle;                          // if value is in the middle:  done

                else if (array[middle] < value)
                        last = middle - 1;                      // toss out the second remaining half
                                                                // of the array and search the first
                else
                        first = middle + 1;                     // toss out the first remaining half of
                                                                // the array and search the second
        }
        return -1;                                              // indicates that value is not in the array
}
```

If you run this program and search for 2, the output indicates that 2 is in the 14^{th} position of the array. Since 2 is in the 14th and 15th position, we see that the binary search found the first occurrence of 2 in this particular data set. However, in the lab you will search for values other than 2 and see that there are other possibilities for which occurrence of a sought value is found.

Sorting Algorithms

We have just seen how to search an array for a specific piece of data; however what if we do not like the order in which the data is stored in the array? For example, if a collection of numerical values is not in order, we might desire them to be ordered so that we can use a binary search to find a particular value. Or, if we have a list of names, we may want them put in alphabetical order. To sort data stored in an array, one uses a **sorting algorithm**. In this section we will consider two such algorithms—the bubble sort and the selection sort.

The Bubble Sort

The bubble sort is a simple algorithm used to arrange data in either **ascending** (lowest to highest) or **descending** (highest to lowest) order. To see how this sort works, let us arrange the array below in ascending order.

9	2	0	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

The bubble sort begins by comparing the first two array elements. If `Element 0 > Element 1`, which is true in this case, then these two pieces of data are exchanged.
The array is now the following:

2	9	0	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

Next elements 1 and 2 are compared. Since `Element 1 > Element 2`, another exchange occurs:

2	0	9	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

Now elements 2 and 3 are compared. Since 9 < 11, there is no exchange at this step.
Next elements 3 and 4 are compared and exchanged:

2	0	9	5	11
Element 0	Element 1	Element 2	Element 3	Element 4

At this point we are at the end of the array. Note that the largest value is now in the last position of the array. Now we go back to the beginning of the array and repeat the entire process over again. Elements 0 and 1 are compared. Since 2 > 0, an exchange occurs:

0	2	9	5	11
Element 0	Element 1	Element 2	Element 3	Element 4

Next elements 1 and 2 are compared. Since 2 < 9, no swap occurs. However, when we compare elements 2 and 3 we find that 9 > 5 and so they are exchanged. Since `Element 4` contains the largest value (from the previous pass), we do not need to make any more comparisons in this pass.

The final result is:

0	2	5	9	11
Element 0	Element 1	Element 2	Element 3	Element 4

The data is now arranged in ascending order and the algorithm terminates. Notice that the larger values seem to rise “like bubbles” to the larger positions of the array as the sort progresses.

We just saw in the previous example how the first pass through the array positioned the largest value at the end of the array. This is always the case. Likewise, the second pass will always position the second to largest value in the second position from the end of the array. The pattern continues for the third pass, fourth pass, and so on until the array is fully sorted. Subsequent passes have one less array element to check than their immediate predecessor.

Example 6.3

```
// This program uses a bubble sort to arrange an array of integers in
// ascending order

#include<iostream>
using namespace std;

// function prototypes

void bubbleSortArray(int [], int);
void displayArray(int[], int);

const int SIZE = 5;

int main()
{
	int values[SIZE] = {9,2,0,11,5};

	cout << "The values before the bubble sort is performed are:" << endl;
	displayArray(values,SIZE);

	bubbleSortArray(values,SIZE);

	cout << "The values after the bubble sort is performed are:" << endl;
	displayArray(values,SIZE);

	return 0;
}
```

```
//*****************************************************************
//                  displayArray
//
// task:            to print the array
// data in:         the array to be printed, the array size
// data out:        none
//
//*****************************************************************

void displayArray(int array[], int elems)                  // function heading
{                                                          // displays the array
        for (int count = 0; count < elems; count++)
                cout << array[count] << "  " << endl;
}

//*****************************************************************
//                  bubbleSortArray
//
// task:            to sort values of an array in ascending order
// data in:         the array, the array size
// data out:        the sorted array
//
//*****************************************************************

void bubbleSortArray(int array[], int elems)
{
        bool swap;
        int temp;
        int bottom = elems - 1;           // bottom indicates the end part of the
                                          // array where the largest values have
                                           // settled in order
do
        {
                swap = false;
                for (int count = 0; count < bottom; count++)
                {
                        if (array[count] > array[count+1])
                        {                 // the next three lines do a swap
                          temp = array[count];
                          array[count] = array[count+1];
                          array[count+1] = temp;
                          swap = true; // indicates that a swap occurred
                        }
                }
        bottom--;     // bottom is decremented by 1 since each pass through
                      // the array adds one more value that is set in order
```

```
        } while(swap != false);
                        // loop repeats until a pass through the array with
                        // no swaps occurs
}
```

While the bubble sort algorithm is fairly simple, it is inefficient for large arrays since data values only move one at a time.

The Selection Sort

A generally more efficient algorithm for large arrays is the **selection sort**. As before, let us assume that we want to arrange numerical data in ascending order. The idea of the selection sort algorithm is to first locate the smallest value in the array and move that value to the beginning of the array (i.e., position 0). Then the next smallest element is located and put in the second position (i.e., position 1). This process continues until all the data is ordered. An advantage of the selection sort is that for *n* data elements at most *n*-1 moves are required. The disadvantage is that *n(n-1)/2* comparisons are always required. To see how this sort works, let us consider the array we arranged using the bubble sort:

9	2	0	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

First the smallest value is located. It is 0, so the contents of `Element 0` and `Element 2` are swapped:

0	2	9	11	5
Element 0	Element 1	Element 2	Element 3	Element 4

Next we look for the second smallest value. The important point to note here is that we do not need to check `Element 0` again since we know it already contains the smallest data value. So the sort starts looking at `Element 1`. We see that the second smallest value is 2, which is already in `Element 1`. Starting at `Element 2` we see that 5 is the smallest of the remaining values. Thus the contents of `Element 2` and `Element 4` are swapped:

0	2	5	11	9
Element 0	Element 1	Element 2	Element 3	Element 4

Finally, the contents of `Element 3` and `Element 4` are compared. Since 11 > 9, the contents are swapped leaving the array ordered as desired:

0	2	5	9	11
Element 0	Element 1	Element 2	Element 3	Element 4

Example 6.4

```
// This program uses a selection sort to arrange an array of integers in
// ascending order

include<iostream>
using namespace std;

// function prototypes

void selectionSortArray(int [], int);
void displayArray(int[], int);
const int SIZE = 5;

int main()
{
        int values[SIZE] = {9,2,0,11,5};

        cout << "The values before the selection sort is performed are:" << endl;
        displayArray(values,SIZE);

        selectionSortArray(values,SIZE);
        cout << "The values after the selection sort is performed are:" << endl;
        displayArray(values,SIZE);

        return 0;
}

//*********************************************************************
//                  displayArray
//
// task:          to print the array
// data in:       the array to be printed, the array size
// data out:      none
//
//*********************************************************************
```

```
void displayArray(int array[], int elems)    // function heading
{                                            // Displays array
        for (int count = 0; count < elems; count++)
                cout << array[count] << "  ";
    cout << endl;
}
//*********************************************************************
//                 selectionSortArray
//
// task:          to sort values of an array in ascending order
// data in:       the array, the array size
// data out:      the sorted array
//
//*********************************************************************

void selectionSortArray(int array[], int elems)
{
        int seek;                       // array position currently being put in order
        int minCount;                   // location of smallest value found
        int minValue;                   // holds the smallest value found

        for (seek = 0; seek < (elems-1);seek++)  // outer loop performs the swap
                                                 // and then increments seek
        {
                minCount = seek;
                minValue = array[seek];
                for(int index = seek + 1; index < elems; index++)
                {
                        // inner loop searches through array starting at array[seek] searching
                        // for the smallest value. When the value is found, the subscript is
                        // stored in minCount. The value is stored in minValue.
                    if(array[index] < minValue)
                    {
                            minValue = array[index];
                            minCount = index;
                    }
                }

                        // the following two statements exchange the value of the element currently
                        // needing the smallest value found in the pass(indicated by seek) with the
                        // smallest value found (located in minValue)
                array[minCount] = array[seek];
                array[seek] = minValue;

        }
}
```

Lab 6: Searching and Sorting Arrays

The purpose of this lab is to give students practice in working with linear & binary search and bubble and selection sort routines.

The lab begins with a quiz on the material covered in this lesson.

Pre-lab Quiz
Fill-in-the-Blank Questions

1. The advantage of a linear search is that it is _____.

2. The disadvantage of a linear search is that it is _____.

3. The advantage of a binary search over a linear search is that a binary search is_____ .

4. An advantage of a linear search over a binary search is that the data must be_____ for a binary search.

5. After 3 passes of a binary search, approximately what fraction of the original array still needs to be searched (assuming the desired data has not been found)? _____

6. While the _____sort algorithm is conceptually simple, it can be inefficient for large arrays because data values only move one at a time.

7. An advantage of the _____sort is that, for an array of size *n*, at most $n - 1$ moves are required.

8. Use the bubble sort on the array below and construct the first 3 steps that actually make changes. (Assume the sort is from smallest to largest).

19	**-4**	**91**	**0**	**-17**
Element 0	Element 1	Element 2	Element 3	Element 4

Element 0	Element 1	Element 2	Element 3	Element 4

Element 0	Element 1	Element 2	Element 3	Element 4

Element 0	Element 1	Element 2	Element 3	Element 4

9. Use the selection sort on the array below and construct the first 3 steps that actually make changes. (Assume the sort if from smallest to largest).

19	**-4**	**91**	**0**	**-17**
Element 0	Element 1	Element 2	Element 3	Element 4

Element 0	Element 1	Element 2	Element 3	Element 4

Element 0	Element 1	Element 2	Element 3	Element 4

Element 0	Element 1	Element 2	Element 3	Element 4

Problem 6.1

Bring in program `linear_search` from the Lab 6 folder. The code is as follows:

```
// This program performs a linear search on a character array

// Place Your Name Here

#include<iostream>
using namespace std;

int searchList( char[], int, char); // function prototype
const int SIZE = 8;

int main()
{
        char word[SIZE] = "Harpoon";
        int found;
        char ch;

        cout << "Enter a letter to search for:" << endl;
        cin >> ch;

        found = searchList(word, SIZE, ch);
        if (found == -1)
                cout << "The letter " << ch
         << " was not found in the list" << endl;
        else
```

```
                cout << "The letter " << ch <<" is in the " << found + 1
                    << " position of the list" << endl;

        return 0;

}


//*****************************************************************
//                                  searchList
//
// task:                This searches an array for a particular value
// data in:              List of values in an array, the number of
//                       elements in the array, and the value searched for
//                      in the array
// data returned:       Position in the array of the value or -1 if value
//                       not found
//
//*****************************************************************

int searchList( char List[], int numElems, char value)
{
        for (int count = 0;count <= numElems; count++)
        {
                if (List[count] == value)
               // each array entry is checked to see if it contains
                     // the desired value.
                  return count;
              // if the desired value is found, the array subscript
                                // count is returned to indicate the location in the array
        }
        return -1;          // if the value is not found, -1 is returned
}
```

Exercise 1: Re-write this program so that it searches an array of integers rather than characters. Search the integer array `nums[8] =`

3	6	-19	5	5	0	-2	99

for several different integers. Make sure you try integers that are in the array and others that are not. What happens if you search for 5?

Problem 6.2

Bring in program `binary_search.cpp` from the Lab 5 folder. The code is as follows:

```
// This program demonstrates a Binary Search

//PLACE YOUR NAME HERE

#include<iostream>
using namespace std;

int binarySearch(int [], int, int);  // function prototype

const int SIZE = 16;

int main()
{
        int found, value;
        int array[] = {34,19,19,18,17,13,12,12,12,11,9,5,3,2,2,0}; // array to be searched

        cout << "Enter an integer to search for:" << endl;
        cin >> value;

        found = binarySearch(array, SIZE, value); //function call to perform the binary search
                                        //on array looking for an occurrence of value
        if (found == -1)
                cout << "The value " << value << " is not in the list" << endl;
        else
        {
                cout << "The value " << value << " is in position number "
                    << found + 1 << " of the list" << endl;
        }
        return 0;
}
//****************************************************************
//                                        binarySearch
//
// task:                        This searches an array for a particular value
// data in:                     List of values in an ordered array, the number of
//                              elements in the array, and the value searched for
//                              in the array
// data returned:               Position in the array of the value or -1 if value
//                               not found
//****************************************************************
int binarySearch(int array[],int numElems,int value) //function heading
```

```
{
        int first = 0;                          // First element of list
        int last = numElems - 1;                // last element of the list
        int middle;                             // variable containing the current
                                                // middle value of the list

        while (first <= last)
        {
                middle = first + (last - first) / 2;

        if (array[middle] == value)
                return middle;                  // if value is in the middle, we are done

        else if (array[middle]<value)
                last = middle - 1;              // toss out the second remaining half of
                                                // the array and search the first
        else
                first = middle + 1;             // toss out the first remaining half of
                                                // the array and search the second
        }

        return -1;                              // indicates that value is not in the array
}
```

Exercise 1: The variable `middle` is declared as an integer. The program contains the assignment statement `middle=first+(last-first)/2`. Is the right side of this statement necessarily an integer in computer memory? Explain how the `middle` value is determined by the computer. How does this line of code affect the logic of the program? Remember that `first, last`, and `middle` refer to the array positions, not the values stored in those array positions.

Exercise 2: Search the array in the program above for 19 and then 12. Record what the output is in each case. _____

Notice that both 19 and 12 are repeated in the array.

Which occurrence of 19 did the search find?_____

Which occurrence of 12 did the search find? _____

Explain the difference.

Exercise 3: Modify the program to search an array that is in ascending order. Make sure to alter the array initialization.

Problem 6.3

Bring in either the program `bubble_sort.cpp` or `selection_sort.cpp` from the Lab 6 folder. The codes for both are given below.

```
// This program uses a bubble sort to arrange an array of integers in
// ascending order

#include<iostream>
using namespace std;

// PLACE NAME HERE

// function prototypes

void bubbleSortArray(int [], int);
void displayArray(int[], int);

const int SIZE = 5;

int main()
{
	int values[SIZE] = {9,2,0,11,5};

	cout << "The values before the bubble sort is performed are:" << endl;
	displayArray(values,SIZE);

	bubbleSortArray(values,SIZE);

	cout << "The values after the bubble sort is performed are:" << endl;
	displayArray(values,SIZE);

	return 0;
}
//*****************************************************************
//                          displayArray
//
// task:              to print the array
// data in:           the array to be printed, the array size
// data out:          none
//
//*****************************************************************

void displayArray(int array[], int elems)    // function heading
```

```
{                                                       // displays the array
        for (int count = 0; count < elems; count++)
                cout << array[count] << "  " << endl;
}

//*****************************************************************
//                          bubbleSortArray
//
// task:                to sort values of an array in ascending order
// data in:             the array, the array size
// data out:            the sorted array
//
//*****************************************************************


void bubbleSortArray(int array[], int elems)
{
        bool swap;
        int temp;
        int bottom = elems - 1;     // bottom indicates the end part of the
                                    // array where the largest values have
                                    // settled in order
        do
        {
                swap = false;
                for (int count = 0; count < bottom; count++)
                {
                        if (array[count] > array[count+1])
                        {               // the next three lines do a swap
                          temp = array[count];
                          array[count] = array[count+1];
                          array[count+1] = temp;
                          swap = true; // indicates that a swap occurred
                        }
                }
        bottom--;     // bottom is decremented by 1 since each pass through
                      // the array adds one more value that is set in order

        } while(swap != false);
          // loop repeats until a pass through the array with  no swaps occurs
}
```

Selection Sort

```
// This program uses a selection sort to arrange an array of integers in
// ascending order

//PLACE NAME HERE

#include<iostream>
using namespace std;

// function prototypes

void selectionSortArray(int [], int);
void displayArray(int[], int);
const int SIZE = 5;

int main()
{
        int values[SIZE] = {9,2,0,11,5};

        cout << "The values before the selection sort is performed are:" << endl;
        displayArray(values,SIZE);

        selectionSortArray(values,SIZE);
        cout << "The values after the selection sort is performed are:" << endl;
        displayArray(values,SIZE);

        return 0;
}

//*********************************************************************
//                              displayArray
//
// task:                 to print the array
// data in:              the array to be printed, the array size
// data out:             none
//
//*********************************************************************

void displayArray(int array[], int elems)    // function heading
{                                                       // Displays array
        for (int count = 0; count < elems; count++)
                cout << array[count] << "  ";
    cout << endl;
}
```

```
//*****************************************************************
//                          selectionSortArray
//
// task:                to sort values of an array in ascending order
// data in:             the array, the array size
// data out:            the sorted array
//
//*****************************************************************

void selectionSortArray(int array[], int elems)
{
        int seek;      //array position currently being put in order
        int minCount;  //location of smallest value found
        int minValue;  //holds the smallest value found

        for (seek = 0; seek < (elems-1);seek++)  // outer loop performs the swap
                                                 // and then increments seek
        {
                minCount = seek;
                minValue = array[seek];
                for(int index = seek + 1; index < elems; index++)
                {
                        // inner loop searches through array  starting at array[seek] searching
                        // for the smallest value. When the value is found, the subscript is
                        // stored in minCount. The value is stored in minValue.

                        if(array[index] < minValue)
                        {
                                minValue = array[index];
                                minCount = index;
                         }
                }

                        // the following two statements exchange the value of the
                        // element currently needing the smallest value found in the
                        /  pass(indicated by seek) with the smallest value found
                        // (located in minValue)

                array[minCount] = array[seek];
                array[seek] = minValue;

        }
}
```

Exercise 1: Re-write the sort program you chose so that it orders integers from largest to smallest

rather than smallest to largest.

Exercise 2: Modify your program from Exercise 1 so that it prints the array at each step of the algorithm. Try sorting the array

23	0	45	-3	-78	1	-1	9

by hand using whichever algorithm you chose. Then have your program do the sort. Does the output match what you did by hand?

Student Generated Code Assignments

Student Generated Code Problem 6.4

Write a program that prompts the user to enter the number of elements and the numbers themselves to be placed in an integer array that holds a maximum of 50 elements. The program should then prompt the user for an integer which will be searched fo in the array using a binary search. Make sure to include the following steps along the way:

i) A sort routine must be called before the binary search. You may use either the selection sort or the bubble sort. However, the sort must be implemented in its own function and not in `main`.

ii) Next include a function called by `main` to implement the binary search. The ordered array produced by the sort should be passed to the search routine which returns the location in the sorted array of the sought value, or -1 if the value is not in the array.

iii) Add a value returning function that computes the mean of your data set. Recall that the mean is the sum of the data values divided by the number of pieces of data. Your program should output the size of the array entered, the array as entered by the user, the sorted array, the integer being searched for, the location of that integer in the sorted array (or an appropriate message if it is not in the array), and the mean of the data set.

iv) (Optional) Modify your program so that the data is entered from a file rather than from the keyboard. The first line of the file should be the size of the integer array. The second line should contain the integer searched for in the data set. Finally, the array elements are to start on the third line. Make sure you separate each array element with a space. The output as described in iii) should be sent to a file.

Lesson 6: Summary

Linear Search searches attempts to find some value in an array.

Its positive attributes are
1) it is an easy routine to understand and program
2) if the element is near the beginning of the array it is efficient

Its negative attributes are
1) it is inefficient since most case the searched element is not at the beginning
2) if the searched item is not in the array, it will have searched every element of the array.

Binary Search

Its positive attribute is that it is efficient. It divides the array in half and searches only half of the array and then half f that half etc. until the element is found or the element is not in the array

Its negative attributes is that the array must be ordered and it is a bit more complex than the linear Search.

Bubble Sort Sorts arrange the data in an array in an ordered fashion

The bubble sort (in an ascending routine) finds the largest value then stores it in the last position, it then finds the next to the largest and stores it into the next to the last position etc.

Selection Sort

The selection sort is more efficient for large arrays.

Its positive attribute is that for n data elements at most n-1 moves are required.
Its negative attribute is that n(n-2)/2 comparisons are always required.

Lesson 7: Characters and Strings

Review of Java data types

In your study of Java, you learned that there were two broad categories of data types: primitive and reference.

Primitive types store various types of numbers and are used to build other types. There are eight basic primitive data types in Java.

1)	byte	used to store small integers in the range of -128 to 127
2)	short	used to store medium integers in the range of -32,768 to 32,767
3)	int	used to store normal integers -2,147,483,648 to 2,147,483,647
4)	long	used to store large integers -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
5)	float	used to store real numbers (those with fractional components)
6)	double	used to store more precise real numbers
7)	boolean	used to store just true or false values (takes up only 1 bit)
8)	char	used to store a single character (ASCII) 1 byte.

Reference types

Reference types store references to objects (instances of classes). They are often called class types because they are the names of classes. There are hundreds of reference types. New reference types can be created by building classes, whereas there will always only be eight primitive types.

In Java, programmers often use the data type `String` as if it were a primitive data type, but as we know it is a reference type because it is built as a class and given a name. (This is why in Java the first letter of `String` is upper case.

C++ string data type

In C++ we looked at the following data types: `short, int, long` (all of which were integer data type), `float, double` (numbers with a decimal fractional component), `bool, char` and `string.`

The data type `string` just like its counterpart in Java is a reference data type which means that any variable (object) created from it contains an address of a location to the starting position of the sequence of characters that make up the string. It is a class rather than a primitive data type.

We now explore more detail about the char data type and functions that deal specificially with it.

Character Functions

C++ provides numerous *functions* for character testing. These functions will test a single character and

return either a non-zero value (true) or zero (false). For example, `isdigit` tests a character to see if it is one of the digits between 0 and 9. So `isdigit(7)` returns a non-zero value whereas `isdigit(y)` and `isdigit($)`both return 0. The following program example demonstrates other character functions. Pay attention to the bold letters. Note that the `cctype` header file must be included to use the character functions.

Example 7.1:

```
// This program utilizes several functions for character testing

#include <iostream>
#include <cctype>
using namespace std;

int main()
{
        char input;
        cout << "Please Enter Any Character:" << endl;
        cin >> input;
        cout << "The character entered is " << input << endl << endl;
        cout << "The ASCII code for " << input << " is " << int(input) << endl;

        if ( isalpha(input)) // tests to see if the character is a letter
        {
                cout << "The character is a letter" << endl;

                if (islower(input)) // tests to see if letter is lower case
                        cout << "The letter is lower case" << endl;

                if (isupper(input)) // tests to see if letter is upper case
                        cout << "The letter is upper case" << endl;
        }
        else if (isdigit(input)) // tests to see if character is a digit
                cout << "The character you entered is a digit" << endl;
        else
                cout << "The character entered is neither a letter nor a" << "digit" << endl;
        return 0;
}
```

In the lab for this lesson you will see a more practical application of character testing functions.

Character Case Conversion

The C++ library provides the `toupper` and `tolower` functions for converting the case of a character. `toupper` returns the uppercase equivalent for a letter and `tolower` returns the lower case equivalent. For example, `cout << tolower('F');` causes an `f` to be displayed on the screen. If the letter is already lowercase, then `tolower` will return the value unchanged. Likewise, any non-letter argument

is returned unchanged by `tolower`. It should be clear to you now what `toupper` does to a given character.

While the `toupper` and `tolower` functions are conceptually quite simple, they may not appear to be very useful. However, the following program shows that they do have beneficial applications.

Example 7.2

```
// This program shows how the toupper and tolower functions can be
// applied in a C++ program

#include <iostream >
#include <cctype >
#include <iomanip>

int main()
{
        int week, total, dollars;
        float average;
        char choice;
        cout << fixed << showpoint << setprecision(2);

        do
        {
                total = 0;
                for(week = 1;week <= 4;week++)
                {
                        cout << "How much (to the nearest dollar) did you"
                              << " spend on food during week " <<week <<" ?:" <<endl;
                        cin >> dollars;
                        total = total + dollars;
                }
                average = total / 4.0;

                cout << "Your weekly food bill over the chosen month is $"
                     << average << endl << endl;
                do
                {
                        cout <<"Would you like to find the average for another month?";
                        cout << endl << "Enter Y or N" << endl;
                        cin >> choice;
                } while(toupper(choice) != 'Y' && toupper(choice) != 'N');

        } while (toupper(choice) == 'Y');
        return 0;
}
```

This program prompts the user to input weekly food costs, to the nearest dollar (an integer) for a four-week period. The average weekly total for that month is output. Then the user is asked whether they want to repeat the calculation for a different month. The flow of this program is controlled by a do-while loop. The condition toupper(choice) == 'Y' allows the user to enter 'Y' or 'y' for yes. This makes the program more user friendly than if we just allowed 'Y'. Note the second do-while loop near the end of the program. This loop also utilizes toupper. Can you determine the purpose of this second loop? How would the execution of the program be affected if we removed this loop (but left in the lines between the curly brackets)?

String Constants

We have already talked about the character data type which includes letters, digits, and other special symbols such as $ and @. The data type `string`, as mentioned earlier, is formed by a series of characters. For example, the price "$1.99" and the phrase "one for the road!" are both strings of characters. The phrase contains blank space characters in addition to letters and an exclamation mark. In C++ a string is treated as a sequence of characters stored in consecutive memory locations. The end of the string in memory is marked by the null character `\0`. Do not confuse the null character with a sequence of two characters (i.e., \ and 0). The null character is actually an escape sequence. Its ASCII code is 0. For example, the phrase above is stored in computer memory as

o	n	e		f	o	r		t	h	e		r	o	a	d	\0

A **string constant** is a string enclosed in double quotation marks. For example,

"Learn C++"
"What time is it?"
"Code Word 7dF#c&Q"

are all string constants. When they are stored in the computer's memory, the null character is automatically appended. The string "Please enter a digit" is stored as

P	l	e	a	s	e		e	n	t	e	r		a		d	i	g	i	t	\0

When a string constant is used in C++, it is the memory address that is actually accessed. In the tatement

```
cout << "Please enter a digit";
```

the memory address is passed to the `cout` object. `cout` then displays the consecutive characters until the null character is reached.

Although this is similar to what could be defined as an array of characters it is different. We can, however store a string as an array of characters.

Storing Strings in Arrays

Often we need to access parts of a string rather than the whole string. For instance, we may want to alter characters in a string or even compare two strings. If this is the case, then a string constant is not what we need. Rather, a character array is the appropriate choice. When using character arrays, enough space to hold the null character must be allocated. For example:

```
char last[10];
```

This code declares a 10-element character array called `last`. However, this array can hold no more than 9 characters since a space is reserved for the null character. Consider the following:

```
char name[10];
cout << "Please enter your last name using no more than 9 letters";
cin >> name;
```

If the user enters Symon, then the following will be the contents of the `name` array:

S	y	m	o	n	\0

Recall that the computer actually sees `name` as the beginning address of the array. There is a problem that can arise when using the `cin` object on a character array. `cin` does not "know" that `name` has only 10 elements. If the user enters Newmanouskous after the prompt, then `cin` will write past the end of the array. We can get around this problem by using the `getline` function. If we use

```
cin.getline(last,10)
```

then the computer knows that the maximum length of the string, including the null character, is 10. Consequently, `cin` will read until the user hits `ENTER` or until 9 characters have been read, whichever occurs first. Once the string is in the array, it can be processed character by character.
In the next section we will see a program that uses `cin.getline()`.

Library Functions for Strings

The C++ library provides many functions for testing and manipulating strings. For example, to determine the length of a given string one can use the `strlen` function. The syntax is shown in the following code:

```
char line[40] = "A New Day";
int length;
length = strlen(line);
```

Here `strlen(line)`returns the length of the string including white spaces but not the null character at the end. So the value of `length` is 9. Note this is smaller than the size of the actual array holding the string.

To see why we even need a function such as `strlen`, consider the problem of reading in a string and then writing it backwards. If we only allowed strings of a fixed size, say length 29 for example, then the task would be easy. We simply read the string into an array of size 30 or more. Then write the 29th entry followed by the 28th entry and so on, until we reach the 0th entry. However, what if we wish to

allow the user to input strings of different lengths? Now it is unclear where the end of the string is. Of course, we could search the array until we find the null character and then figure out what position it is in. But this is precisely what the `strlen` function does for us. The following example is a complete program that performs the desired task. Note that we include the cstring directive to use these functions.

Example 7.3:

```
#include <iostream >
#include <cstring.h>                    // necessary to use strlen

int main()
{
        char line[50];
        int length, count = 0;

        cout << "Enter a sentence of no more than 49 characters:\n";
        cin.getline(line,50);

        length = strlen(line);          // strlen returns the length of the string currently stored in line
        cout << "The sentence you entered printed backwards is:\n";

        for(count = length-1; count >= 0; count--)
        {
                cout << line[count];
        }
        cout << endl;
        return 0;
}
```

Sample Run 1:

```
Enter a sentence of no more than 49 characters:
luaP deiruB I
The sentence you entered printed backwards is:
I Buried Paul
```

Sample Run 2:

```
Enter a sentence of no more than 49 characters:
This sentence is too long to hold a mere 49 characters!
The sentence you entered printed backwards is:
arahc 94 erem a dloh ot gnol oot si ecnetnes sihT
```

Another useful function for strings is `strcat`, which concatenates two strings. `strcat(string1,string2)` attaches the contents of `string2` to the end of `string1`. The programmer must make sure that the array containing string1 is large enough to hold the concatenation

of the two strings plus the null character.

Consider the following code:

```
char string1[25] = "Total Eclipse ";   // note the space after the second
                                        // word - strcat does not insert a
                                       // space. The programmer must do this.
char string2[11] = "of the Sun";
cout << string1 << endl;
cout << string2 << endl;
strcat(string1,string2);
cout << string1 << endl;
```

These statements produce the following output:
Total Eclipse
of the Sun
Total Eclipse of the Sun

What would have happened if we had declared `string1` to be a character array of size 20?

There are several other string functions such as `strcpy` (copies the second string to the first string), `strcmp` (compares two strings to see if they are the same or, if not, which string is alphabetically greater than the other), and `strstr` (looks for the occurrence of a string inside of another string).

The get and ignore functions

There are several ways of inputting strings. We could use the standard >> extraction operator for a character array or string class object. However, we know that using `cin >>` skips any leading whitespace (blanks, newlines). It will also stop at the first trailing whitespace character. So, for example, the name "John Wayne" cannot be read as a single string using `cin >>` because of a balnk space between the first and last names. We have already seen the `getline` function which does allow blank spaces to be read and stored. In this section we will introduce the `get` and `ignore` functions, which are also useful for stirng processing.

The get function reads in the next character in the input stream, including whitespace. The syntax is `cin.get(ch);`

Once this function call is made, the next character in the input stream is stored in the variable `ch`. So if we want to input

```
$  X
```

We can use the following:

```
cin.get(firstChar);
cin.get(ch);
cin.get(secondChar);
```

where `firstChar`, `ch` and `secondChar` are all character variables. Note that after the second call to the `get` function, the blank character is stored in the variable `ch`.

The `get` function, like the `getline` function, can also be used to read strings. In this case we need two parameters:

```
cin.get(strName, numChar+1);
```

Here `strName` is a string variable and the integer expression `numChar+1` gives the number of characters that may be reand into `strName`.

Both the `getline` and the `get` functions do not skip leading whitespace characters. The get statement above brings in the next input characters until it either has read `numChar+1` characters or it reaches the newline character `\n`. However, the newline character is not stored in `strName`. The null character is then appended to the end of the string. Since the newline character is not **consumed** (not read by the `get` function), it remains part of the input characters yet to be read!

Example

```
char strName [21];
cin.get (strName, 21);
```

Now suppose we input **John Wayne**
Then "John Wayne" is stored in `strName`.

Now suppose we input **My favorite westerns star John Wayne**

In this case the string "My favorite westerns" is stored in `strName`. It stores only the first 20 characters (it uses the 21st position for the end of string character).

We often work with records from a file that contain character data followed by numeric data. Look at the following data which has a name, hours worked, and pay rate for each record stored on a separate line.

Pay Roll Data

John Brown	7	12.50
Mary Lou Smith	12	15.70
Dominic DeFino	8	15.50

Since names often have imbedded blank spaces, we can use the `get` function to read them. We then use an integer variable to store the number of hours and a floating point variable to store the pay rate. At the end of each line is the `'\n'` character. Note that the end of line character is not consumed by reading the pay rate and, in fact, is the next character to be read when reading the second name from the file. This creates problems. Whenever we need to read through characters in the input steam without storing them, we can use the `ignore` function. This function has two arguments, the first is an integer expression and the second is a character expression. For example, the call

```
cin.ignore(80,'\n');
```

says to skip over the next 80 input characters but stop if a newline character is read. The newline character IS consumed by the ignore function. This use of ignore is often employed to find the end of the current input line.

The following example will read the sample pay roll data from a file called payroll.dat and show the result to the screen. Note that the input file must have names that are no longer than 15 characters and the first 15 posisitons of each line are reserved for the name. The numeric data must be after the 15th position in each line.

Example 7.4

```
#include <fstream>
#include <iostream>
using namespace std;

const int MAXNAME =15;

int main()

{
        if stream inData;

        inData.open("payroll.dat");
        char  name[MAXNAME+1];
        int hoursWorked;
        float payRate;

        inData.get(name, MAXNAME+1) ;  // prime the read
        while (indata)
        {
                inData >> hoursWorked;
                inData >> pay Rate;

                cout << name << endl;
                cout << "Hours Worked "  << hoursWorked << endl;
                cout << "Pay Rate " << payRate << " per hour"
                      <<  endl << endl;

                inData.ignore(80,'\n');
                //  This will ignore up to 80 characters but will stop (ignoring) when
                //  it read the '\n' which is consumed.

                inData.get(name,MAXNAME+1);
        }
```

```
        return 0;
}
```

Summary of types of input for strings:

```
cin >> strName;           // skips leading whitespace.  Stops at the first trailing whitespace
                          // which is not consumed.
cin.get(strName, 21);     // does not skp leading whitespace.  Stops when either 20
                          // characters are read or '\n' is encountered (which is not consumed.)
cin.ignore(200,'\n')      // ignores at most 200 characters but stops if newline (which is
                          // consumed) is encountered
```

Concatenation

Strings can be concatenated with string variables or string literals or character data or character literals. The + is the symbol for string concatenation.

Example

```
name = "Dean";
first = "Hi " +  name;
cout << first;          // Prints Hi Dean
first = first + " DeFino";
cout << first;       //Prints   Hi Dean DeFino
first = first + '!';
cout << first;    //Prints Hi Dean DeFino!
letter='.'
name = name + letter;
cout << name;     //Prints    Dean.
```

length /size is a function that is applied to an variable of type string using dot notation: the name of the string variable, followed by a dot, followed by the name of the function. It is a value returning function, so it is used in an expression. It returns the number of characters in the string.

Example

```
int numberOfChar;
string name = "Dean DeFino";
numberOfChar = name.size();
cout << numberOfChar << endl;      //prints 11    The number of characters in the string.
```

find Looks for its argument in the string to which it is applied and returns the position of that character in the string. NOTE: C++ starts counting at 0 so what we might think of as the 14 position will be position 13 in C++

Example:

```
int pos;
string name = “Dean DeFino”;
pos = name.find(“Fino”);
cout << pos << endl;                    // prints 7 the starting location of Fino in the name string.
```

If the string is not found then a strange value is returned. It is the value npos which is the largest value the function will produce.

substr This has two arguments a beginning location of the string and a length from that location of the string. It will return the string value of the characters within those locations.

Example:

```
string name = “Dean DeFino”;

cout << name.substr(5,3);    //prints DeF    Starts at pos 5 and goes for 3 locations.
```

Lab 7: Characters and Strings

The purpose of the lab is to give practical application in working with string functions, array of characters and character and string Input/Output.

Pre-lab Quiz

1. The code `cout << toupper('b');` causes a _____to be displayed on the screen.

2. The data type of `isalpha('g')` is _____ .

3. After the assignment statement `result = isdigit('$')`, result has the value _____ .

4. The code `cout << tolower('#');` causes a _____to be displayed on the screen.

5. The end of a string is marked in computer memory by the _____.

6. In `cin.getline(name,25)`, the 25 indicates that the user can input at most_____ characters into name.

7. Consider the following:

```
char message[35] = "Like tears in the rain";
int length;
length = strlen(message);
```

Then the value of length is _____.

8. Consider the code

```
char string1[30] = "In the Garden";
char string2[15] = "of Eden";
strcat(string1,string2);
cout << string1;
```

The output for this is _____ .

9. The _____ header file must be included to access the `islower` and `isspace` character functions.

10. In C++, a string constant must be enclosed in _____ whereas a character constant must be enclosed in _____.

Problem 7.1 Character Testing and String Validation

The American Equities investment company offers a wide range of investment opportunities ranging from mutual funds to bonds. Investors can check the value of their portfolio from the American

Equities' web page. Information about personal portfolios is protected via encryption and can only be accessed using a password. The American Equities company requires that a password consist of 8 characters: 5 of which must be letters and the other 3 digits. The letters and digits can be arranged in any order. For example,

rt56AA7q
123actyN
1Lo0Dwa9
myNUM741

are all valid passwords. However, the following are all invalid:

the476NEw // It contains more than 8 characters (also more than 5 letters)
be68moon // It contains less than 3 digits.
$retrn99 // It contains only 2 digits and has an invalid character ('$')

American Equities needs a program for their web page that determines whether or not an entered password is valid. Bring in the american_equities.cpp from the Lab 7 folder which does perform this task. The code is the following:

```
// This program tests a password for the American Equities
// web page to see if the format is correct

// Place Your Name Here

#include <iostream>
#include <cctype>
#include <cstring>

using namespace std;

//function prototypes

bool testPassWord(char[]);
int countLetters(char*);
int countDigits(char*);

int main()
{
        char passWord[20];

        cout << "Enter a password consisting of exactly 5 " << "letters and 3 digits:" << endl;
        cin.getline(passWord,20);

        if (testPassWord(passWord))
                cout << "Please wait - your password is being verified" << endl;
        else
        {
                cout << "Invalid password. Please enter a password "
```

```
                << "with exactly 5 letters and 3 digits" << endl;
        cout << "For example, my37RuN9 is valid" << endl;
    }


    // FILL IN THE CODE THAT WILL CALL countLetters and
    // countDigits and will print to the screen both the number of
    // letters and digits contained in the password.

    return 0;
}


//*************************************************************
//                          testPassWord
//
// task:            determines if the word contained in the
//                  character array passed to it, contains
//                  exactly 5 letters and 3 digits.
// data in:         a word contained in a character array
// data returned:   true if the word contains 5 letters & 3
//                  digits, false otherwise
//
//*************************************************************
bool testPassWord(char custPass[])
{
    int numLetters, numDigits, length;

    length = strlen(custPass);
     numLetters = countLetters(custPass);
    numDigits = countDigits(custPass);
    if (numLetters == 5 && numDigits == 3 && length == 8 )
        return true;
    else
        return false;
}
// the next 2 functions are from Sample Program 10.5
//*************************************************************
//                          countLetters
//
// task:            counts the number of letters (both
//                  capital and lower case in the string
// data in:         a string
// data returned:   the number of letters in the string
//
//*************************************************************
```

```
int countLetters(char *strPtr)
{
	int occurs = 0;

	while(*strPtr != '\0')

	{
		if (isalpha(*strPtr))
			occurs++;
		strPtr++;
	}

	return occurs;
}

//*****************************************************************
//                                  countDigits
//
// task:                counts the number of digitts in the string
// data in:             a string
// data returned:       the number of digits in the string
//
//*****************************************************************
int countDigits(char *strPtr)  // this function counts the  number of digits
{
	int occurs = 0;

	while(*strPtr != '\0')
	{
		if (isdigit(*strPtr))  // isdigit determines if  the character is a digit
			occurs++;
		strPtr++;
	}

	return occurs;
}
```

Exercise 1: Fill in the code in bold and then run the program several times with both valid and invalid passwords. Read through the program and make sure you understand the logic of the code.

Exercise 2: Alter the program so that a valid password consists of 10 characters, 6 of which must be digits and the other 4 letters.

Exercise 3: Adjust your program from Exercise 2 so that only lower case letters are allowed for valid passwords.

Problem 7.2 Case Conversion

Bring in case_convert.cpp from the Lab 7 folder. Note that this is Example 7.2 The code is the following:

```
// This program shows how the toupper and tolower functions can be
// applied in a C++ program

#include <iostream>
#include <cctype>
#include <iomanip>
using namespace std;

int main()
{
        int week, total, dollars;
        float average;
        char choice;

        cout << showpoint << fixed << setprecision(2);
        do
        {
                total = 0;
                for(week = 1;week <= 4;week++)
                {
                        cout << "How much (to the nearest dollar) did you"
                                <<" spend on food during week " << week << " ?:" << endl;
                        cin >> dollars;

                        total = total + dollars;
                }
                average = total / 4.0;
                cout << "Your weekly food bill over the chosen month is $"
                         << average << endl << endl;
                do
                {
                  cout << "Would you like to find the average for another month?";
                  cout << endl << "Enter Y or N" << endl;
                  cin >> choice;
                } while(toupper(choice) != 'Y' && toupper(choice) != 'N');

        } while (toupper(choice) == 'Y');

        return 0;
}
```

Exercise 1: Run the program several times with various inputs.

Exercise 2: Notice the following do-while loop which appears near the end of the program:

```
do
{
      cout << "Would you like to find the average for another month?";
      cout << endl << "Enter Y or N" << endl;
      cin >> choice;
} while(toupper(choice) != 'Y' && toupper(choice) != 'N');
```

How would the execution of the program be different if we removed this loop? Try removing the loop but leave the following lines in the program:

```
cout << "Would you like to find the average for another month?";
cout << endl << "Enter Y or N" << endl;
cin >> choice;
```

Record what happens when you run the new version of the program.

Exercise 3: Alter program `case_convert.cpp` so that it performs the same task but uses `tolower` rather than `toupper`.

Program 7.3 Reading records from a file

Bring in program grades.cpp and grades.txt from the Lab 7 folder. The code as as follows:

```
#include <fstream>
#include <iostream>
using namespace std;

// PLACE YOUR NAME HERE

const MAXNAME = 20;

int main()
{
        ifstream inData;
        inData.open("grades.txt");

        char name[MAXNAME + 1];  // holds student name
        float average;  // holds student average
        inData.get(name,MAXNAME+1);
        while (inData)
        {
                inData >> average;
                // FILL IN THE CODE to print out name and  student average

                // FILL IN THE CODE to complete the while
                // loop so that the rest of the student
```

```
            // names and average are read in properly
        }

        return 0;
}
```

Exercise 1

Fill in the code in bold so that the data is properly read from grades.txt. and the desired output to the screen is as follows:

OUTPUT TO SCREEN		DATA FILE	
Adara Starr	has a(n) 94 average	Adara Starr	94
David Starr	has a(n) 91 average	David Starr	91
Sophia Starr	has a(n) 94 average	Sophia Starr	94
Maria Starr	has a(n) 91 average	Maria Starr	91
Danielle DeFino	has a(n) 94 average	Danielle DeFino	94
Dominic DeFino	has a(n) 98 average	Dominic DeFino	98
McKenna DeFino	has a(n) 92 average	McKenna DeFino	92
Taylor McIntire	has a(n) 99 average	Taylor McIntire	99
Torrie McIntire	has a(n) 91 average	Torrie McIntire	92
Emily Garrett	has a(n) 97 average	Emily Garrett	97
Lauren Garrett	has a(n) 92 average	Lauren Garrett	92
Marlene Starr	has a(n) 83 average	Marlene Starr	83
Donald DeFino	has a(n) 73 average	Donald DeFino	73

Student Generated Code Assignments

Student Generated Code Problem 7.4 Using getline() and get()

Exercise 1: Write a short program called `readata.cpp` that defines a character array called `lastName` that contains 10 characters. Prompt the user to enter their last name using no more than 9 characters. The program should then read the name into `lastName` and then output the name back to the screen with an appropriate message.

Exercise 2: Once the program in Exercise 1 is complete, run the program and enter the name **Newmanouskous** at the prompt. What, if anything, happens? (Note that the results could vary depending on your system).

Exercise 3: Re-write the program above using the `getline()` function (and only allowing 9 characters to be input). As before, use the character array last consisting of 10 elements. Run your new program and enter **Newmanouskous** at the prompt. What is the output?

Student Generated Code Program 7.5 String Functions—strcat

Consider the following code:

```
char string1[25] =" Total Eclipse ";
char string2[11] = "of the Sun";
cout << string1 << endl;
cout << string2 << endl;
strcat(string1,string2);
cout << string1 << endl;
```

Exercise 1: Write a complete program including the above code that outputs the concatenation of string1 and string2. Run the program and record the result.

Exercise 2: Alter the program in Exercise 1 so that string1 contains 20 characters rather than 25. Run the program. What happens?

Student Generated Code Program 7.6 Palindrome

A **palindrome** is a string of characters that reads the same forwards as backwards.

For example, the following are both palindromes:
1457887541 madam

Exercise 1: Write a program that prompts the user to input a string of a size 50 characters or less. Your program should then determine whether or not the entered string is a palindrome. A message should be displayed to the user informing them whether or not their string is a palindrome.

Exercise 2: The `strcmp(string1, string2)` function compares `string1` to `string2`. It is a value returning function that returns a negative integer if `string1 < string2`, 0 if `string1 == string2`, and a positive integer if `string1 > string2`. Write a program that reads two names (last name first followed by a comma followed by the first name) and then prints them in alphabetical order. The two names should be stored in separate character arrays holding a maximum of 25 characters eachl. Use the `strcmp()` function to make the comparison of the two names. Remember tha ‘a’ < ‘b’, ‘b’ < ‘c’, etc. Be sure to include the proper header file to use strcmp().

Sample Run 1:

Please input the first name
Brown, George

Please input the second name
Adams, Sally

The names are as follows:

Adams, Sally
Brown, George

Sample Run 2:

Please input the first name
Brown, George

Please input the second name
Brown, George

The names are as follows:

Brown, George
Brown, George
The names are the same

Student Generated Code Program 7.7 Counting consonants

Write a program that determines how many consonants are in an entered string of 50 characters or less. Output the entered string and the number of consonants in the string.

Lesson 7: Summary

Character Functions

toupper(character) is a value returning function that returns the upper case value of character.
tolower(character) is a value returning function that returns the lower case value of character.
isdigit(y) is a value returning function that returns a 1 if y is a digit 0 -9 and a 0 if it is not.
isalpha(y) is a value returning function that returns a 1 if y is a letter and a 0 if it is not
islower(y) is a value returning function that returns a 1 if y is a lower case letter and a 0 if not.
isupper(y) is a value returning function that returns a 1 if y is an upper case letter and a 0 if not.

A **string constant** is a string enclosed in double quotation marks. When they are stored in the computer's memory, the null character is automatically appended. The string "Please enter a digit" is stored as

P	l	e	a	s	e		e	n	t	e	r		a		d	i	g	i	t	\0

Strings can by represented as an array of characters.

char last[10] ; This declares an array of characters that can hold at most 9 characters. The 10th element is reserved for the null character that indicates the end of the string.

String Functions

find Looks for its argument in the string to which it is applied and returns the position of that character in the string. NOTE: C++ starts counting at 0 so what we might think of as the 14 position will be position 13 in C++

```
int pos;
string name = "Dean DeFino";
pos = name.find("Fino");
cout << pos << endl;                    // prints 7 the starting location of Fino in the name string.
```

If the string is not found then a strange value is returned. It is the value npos which is the largest value the function will produce.

substr This has two arguments a beginning location of the string and a length from that location of the string. It will return the string value of the characters within those locations.

```
string name = "Dean DeFino";
cout << name.substr(5,3);    //prints DeF    Starts at pos 5 and goes for 3 locations.
```

```
cin >> strName;              // skips leading whitespace.  Stops at the first trailing whitespace
                             // which is not consumed.
cin.get(strName, 21);        // does not skp leading whitespace.  Stops when either 20
                            // characters are read or '\n' is encountered (which is not consumed.)
cin.ignore(200,'\n')        // ignores at most 200 characters but stops if newline (which is
                           // consumed) is encountered
```

Lesson 8: Structures

Introduction

In the last two lessons we concentrated on the concept of arrays which are useful in many applications; however as practical as they may be, they do have several limitations, two of which will be addressed in this course. One limitation to an array is that its size is static. That means that the number of elements contained in an array must be declared at the start and cannot be changed dynamically. This creates the problem of wasted memory space. A robust program is one that can handle a variety of related problems. If we don't know the amount of elements that will be read into an array we must declare our array to be a maximum size, which in most cases will result in wasted memory space. For example, if we are reading grades from a class we need to know the maximum size of students for any class that would be used in the program. Writing a program for a university that has class sizes ranging from 10 to 150 would necessitate the program to declare an array of 150 elements even if in most cases the class size is under 40. A second limitation is that all the elements of the array must be of the same data type.

In this lesson we address the second limitation by introduction structures : the idea of storing memory locations of different data types under a single name. We address the first problem in the lesson on pointers.

Records

Many real world applications deal with the concept of a record. Look at the following information:

Name	SSNumber	Average Grade	Letter Grade
Brown, Scott	187482019	75.4	C
Davidson, William	839201839	94.7	A
…….			

The logical grouping of this information (grouping that makes sense for applications) is the row of data. Each row contains information about a particular student. But the elements of that group are of different data types. The name would have a string data type, while the SSNumber would be a long integer, the average grade a float or double data type and the letter grade a character data type. An array would not be able to contain all that information since an array must contain homogenous data types. A **record** is a group of fields (each of which can be of different data types) that are grouped logically together. In C++ they can be declared as **structures.**

Abstract Data Types

So far we have learned of data types such as `float, int, char`, etc. In some applications the programmer needs to create their own data type. A user defined data type is often called an **abstract data type (ADT)**. The programmer must decide which values are valid for the data type and which

operations may be performed on the data type. It may even be necessary for the programmer to design new operations to be applied to the data. We study this style of programming in greater deal in a later lesson.

As an example, suppose you want to create a program to simulate a calendar. The program may contain the following ADTs: `year, month`, and `day`. Note that `month` could take on values January, February, . . . , December or even 1,2, . . . ,12 depending on the wishes of the programmer. Likewise, the range of values for `day` could be Monday, Tuesday, . . . , Sunday or even 1,2, . . . ,7. There is much more flexibility in the choice of allowable values for `year`. If the programmer is thinking short term they may wish to restrict `year` to the range 2010-2020. Of course there are many other possibilities.

A **structured data type** is a data type that is a collection of components whose organization is characterized by the method used to access individual components. The allowable operations on a structured data type include the storage and retrieval of individual components.

From this definition we can see that an array is a structured data type.

Structures

Another example of a C++ structured data type which is also an abstract data type is the **structure**. Like arrays, structures allow the programmer to group data together. However, unlike an array, structures allow you to group together items of *different* data types. To see how this could be useful in practice, consider what a student must do to register for a college course. Typically, one obtains the current list of available courses and then selects the desired course or courses. The following is an example of a course you may choose:

CHEM 310 Physical Chemistry 4 Credits

Note that there are four items related to this course: the course discipline (CHEM), the course number (310), the course title (Physical Chemistry), and the number of credit hours (4). We could define variables as follows:

Variable Defintion	**Information Held**
char discipline[5]	4-letter abbreviation for discipline
int courseNumber	Integer valued course number
char courseTitle[21]	First 20 characters of course title
short credits	Number of credit hours

All of these variables are related because they can hold information about the same course. We can package these together by creating a structure. Here is the declaration:

```
struct course
{
        char discipline[5];
        int courseNumber;
        char courseTitle[21];
        short credits;
}; //note the semi-colon here
```

The **tag** is the name of the structure, course in this case. The tag is used like a data type name. Inside the braces we have the variable declarations that are the **members** of the structure. So the code above declares a structure named course which has four members: `discipline, courseNumber, courseTitle`, and `credits`.

The programmer needs to realize that the structure declaration above does not define a variable. Rather it lets the compiler know what a `course` structure is composed of. That is, the declaration creates a new data type (an abstract data type) called `course`. We can now define variables of type `course` as follows:

```
course pChem;
course colonialHist;
```

Both `pChem` and `colonialHist` will contain the four members listed above. We could have also declared these two structure variables on a single line:

```
course pChem, colonialHist;
```

Both `pChem` and `colonialHist` are called **instances** of the `course` structure. In other words, they are both user defined variables that exist in computer memory. Each structure variable contains the four structure members.

Access to Structure Members

Certainly the programmer will need to be able to assign the members values and also keep track of what values the members have. A member selector is an expression used to access components of a struct variable. It is formed by using the struct variable name and the member name, separated by a dot which is called the **dot operator**. Consider the following syntax:

```
colonialHist.credits = 3;
```

In this statement the number 3 is assigned to the `credits` member of colonialHist. The dot operator is used to connect the member name to the structure variable it belongs to.

Now let us put all of these ideas together into a program. Example 8.1 below uses the `course` structure just described. This interactive program allows a student to add requested courses and keeps track of the number of credit hours for which they have enrolled. The execution is controlled by a `do-while` loop.

Example 8.1

```
#include <iostream>
#include <cctype>
using namespace std;

//  This program demonstrates the use of structures

const int MAXDISCIPLINE = 4;
const int MAXCOURSE =20;

// structure declaration

struct course
{
        char discipline[MAXDISCIPLINE+1];
        int courseNumber;
        char courseTitle[MAXCOURSE+1];
        short credits;
};
int main()
{
course nextclass; // next class is a course structure
int numCredits = 0;
char addclass;
do
{

        cout << "Please enter course discipline area: ";
        cin >> nextclass.discipline;
        cout << "Please enter the course number: ";
        cin >> nextclass.courseNumber;
        cout << "Please enter the course title: ";
        cin.ignore(); // necessary for the next line
        cin.getline(nextclass.courseTitle, MAXCOURSE+1);
        // we add an extra space to read the end of line character
        // use gertline because course title may have a blank space
        cout << "Please enter the number of credit hours:  " ;
        cin >> nextclass.credits;

        numCredits = numCredits + nextclass.credits;

        // output the selected course and pertinent information

        cout << "You have been registered for the following: " << endl;
        cout << nextclass.discipline << " " << nextclass.courseNumber
              <<  " "  << nextclass.courseTitle << " " <<  nextclass.credits <<  " credits" << endl;
```

```
        cout << "Would you like to add another class? (Y/N)" << endl;
        cin >> addclass;

} while(toupper(addclass == 'Y'));

cout << "The total number of credit hours registered for is: "
     << numCredits << endl;

return 0;
}
```

Make sure that you understand the logic of this program and, in particular, how structures are used. Notice the line at the end of the `while` loop that reads

```
while(toupper(addclass) == 'Y');
```

What do you think the purpose of `toupper` is?

As a second example, suppose we would like a simple program that computes the area and circumference of two circles input by the user. Although we can easily do this by using previously developed techniques, let us see how this can be done using structures. We will also determine which circle's center is further from the origin.

Example 8.2

```
`
#include <iostream>
#include <math>       // necessary for the pow function
#include <iomanip>
using namespace std;

struct circle                    // declares the structure circle. This structure has 6 members.
{
        double centerX;                 // x coordinate of center
        double centerY;                 // y coordinate of center
        double  radius;
        double area;
        double circumference;
        double distance_from_origin; // distance of center from origin
};

const double PI = 3.14159;
int main()

{
        circle circ1, circ2; // defines 2 circle structure variables
        cout << "Please enter the radius of the first circle: ";
```

```
cin >> circ1.radius;
cout << "\nPlease enter the x-coordinate of the center: ";
cin >> circ1.centerX;
cout << "\nPlease enter the y-coordinate of the center: ";
cin >> circ1.centerY;

circ1.area = PI * pow(circ1.radius,2.0);
circ1.circumference = 2 * PI * circ1.radius;
circ1.distance_from_origin = sqrt(pow(circ1.centerX,2.0) + pow(circ1.centerY,2.0));
cout << endl << endl;
cout << "Please enter the radius of the second circle: ";
cin >> circ2.radius;
cout << "\nPlease enter the x-coordinate of the center: ";
cin >> circ2.centerX;
cout << "\nPlease enter the y-coordinate of the center: ";
cin >> circ2.centerY;

circ2.area = PI * pow(circ2.radius,2.0);
circ2.circumference = 2 * PI * circ2.radius;
circ2.distance_from_origin = sqrt(pow(circ2.centerX,2.0) +pow(circ2.centerY,2.0));
cout << endl << endl;

// This next section determines which circle's center is closer  to the origin

if (circ1.distance_from_origin > circ2.distance_from_origin)
{
        cout << "The first circle is further from the origin" << endl << endl;
}
else if (circ1.distance_from_origin < circ2.distance_from_origin)
{
        cout << "The first circle is closer to the origin" << endl << endl;
}
else
        cout << "The two circles are equidistant from the origin";
cout << endl << endl;

cout << setprecision(2) << fixed << showpoint;

cout << "The area of the first circle is :";
cout << circ1.area << endl;
cout << "The circumference of the first circle is: ";
cout << circ1.circumference << endl << endl;

cout << "The area of the second circle is :";
cout << circ2.area<<endl;
cout << "The circumference of the second circle is: ";
```

```
        cout << circ2.circumference << endl << endl;
        return 0;
}
```

Arrays of Structures

In the previous sample program we were interested in two instances of the circle structure. What if we need a much larger number, say 100, instances of this structure? Rather than define each one separately, we could use an **array of structures**. An array of structures is defined just like any other array. For example suppose we already have the following structure declaration in our program:

```
struct circle                   // declares the structure circle. This structure has 6 members.
{
        double centerX;                 // x coordinate of center
        double centerY;                 // y coordinate of center
        double  radius;
        double area;
        double circumference;
        double distance_from_origin; // distance of center from origin
};
```

Then the following statement declares an array, `circn`, which has 100 elements. Each of these elements is a `circle` structure variable:

```
circle circn[100];
```

Like the arrays encountered in previous lessons, you can access an array element using its subscript. So `circn[0]` is the first structure in the array, `circn[1]` is the second and so on. The last structure in the array is `circn[99]`. To access a member of one of these array elements, we still use the dot operator. For instance, `circn[9].circumference` gives the circumference member of `circn[9]`. If we wanted to display the center and distance from the origin of the first 30 circles we could use the following:

```
for (int count = 0; count < 30; count++)
{
      cout << circn[count].centerX << endl;
      cout << circn[count].centerY << endl;
      cout << circn[count].distance_from_origin;
}
```

When studying arrays you may have seen two-dimensional arrays which allow one to have “a collection of collections” of data. An array of structures allows one to do the same thing. However, we have already noted that structures permit you to group together items of different data type, whereas arrays do not. So using an array of structures can sometimes be used where a two-dimensional array cannot.

We can use the typedef statement in setting up structures and other ADTs.

Look at the following example:

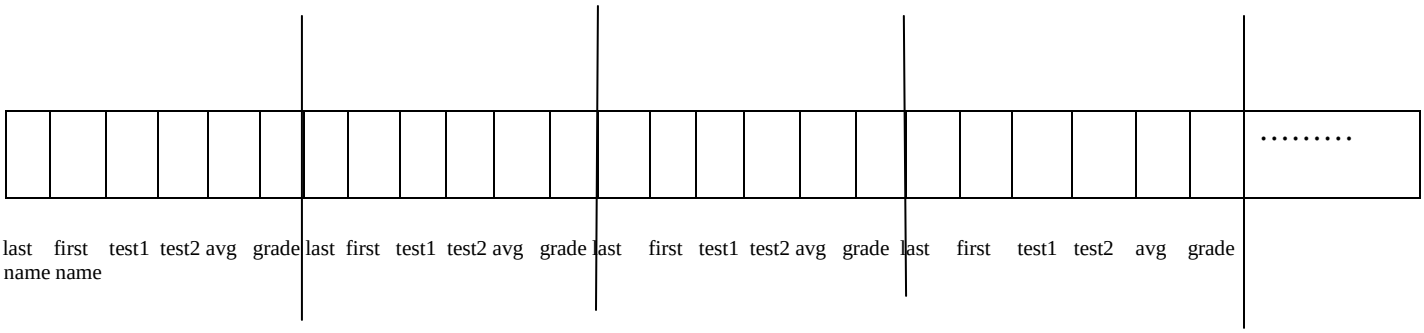
Example 8.3

```
const MAX_STUDENTS = 200;

struct Student_data_type
{
  string      lastname;
  string      firstname;
  int         test1;
  int         test2;
  float       average;
  char        grade;
}  ;

typedef   Student_data_type   Student_array_type[MAX_STUDENTS];

int main()
{
        Student_array_type    student;
        int pos = 0;

        indata >> student[pos].lastname;
        while(indata)
{
        indata >> student[pos].firstname >> student[pos].test1 >>  student[pos].test2;

        student[pos].average = (student[pos].test1 + student[pos].test2) / 2;
        // determine letter grade
              // instructions go here that will determine the letter grade

        //output information
        // instructions go here that will output the information
        pos++;
        indata >> student[pos].lastname
}
```

Questions What are the following data types?

student[pos].test 1 integer
student[pos] Student_data_type
student Student_array_type

}



Initializing Structures

We have already seen numerous examples of initializing variables and arrays at the time of their definition. Members of structures can also be initialized when they are declared. Suppose we have the following structure declaration in our program:

```
struct course
{
        char discipline[5];
        int courseNumber;
        char courseTitle[20];
        short credits;
};
```

A structure variable `colonialHist` can be declared and initialized:

```
course colonialHist = {"HIST",302,"Colonial History",3};
```

The values in this list are assigned to `course`'s members in the order they appear. Thus, the string "`HIST`" is assigned to `colonialHist.discipline`, the integer `302` is assigned to `colonialHist.courseNumber`, the string "`Colonial History`" is assigned to `colonialHist.courseTitle`, and the short value `3` is assigned to `colonialHist.credits`. It is not necessary to initialize all the members of a structure variable. For example, we could initialize just the first member:

```
course colonialHist = {"HIST"};
```

This statement leaves the last three members uninitialized. We could also initialize only the first two members:

```
course colonialHist = {"HIST",302};
```

There is one important rule, however, when initializing structure members. If one structure member is left uninitialized, then all structure members that follow must be uninitialized. In other words, we cannot skip members of a structure during the initialization process.

It is also worth pointing out that you cannot initialize a structure member in the declaration of the structure. The following is an illegal declaration:

```
// illegal structure declaration
struct course
{
        char discipline[5] = "HIST"; // illegal
        int courseNumber = 302; // illegal
        char courseTitle[20] = "Colonial History"; // illegal
        short credits = 3; // illegal
};
```

If we recall what a structure declaration does, it is clear why the above code is illegal. A structure declaration simply lets the compiler know what a structure is composed of. That is, the declaration creates a new data type (called `course` in this case). So the structure declaration does not define any variables. Hence there is nothing that can be initialized there.

Hierarchical (Nested) Structures

Often it is useful to nest one structure inside of another structure. Consider the following:

Example 8.4

```
#include <iostream >
#include <iomanip>
#include <cmath>
using namespace std;

struct center_struct
{
        double x;                       // x coordinate of center
        double y;                       // y coordinate of center
};

struct circle
{
        double radius;
        doublet area;
        double circumference;
        center_struct coordinate;
};
const double PI = 3.14159;
int main()
{
        circle circ1, circ2;            // declares 2 circle structure variables
        cout << "Please enter the radius of the first circle: ";
        cin >> circ1.radius;
        cout << "\nPlease enter the x-coordinate of the center: ";
```

```
        cin >> circ1.coordinate.x;
        cout << "\nPlease enter the y-coordinate of the center: ";
        cin >> circ1.coordinate.y;

        circ1.area = PI * pow(circ1.radius,2.0);
        circ1.circumference = 2 * PI * circ1.radius;
        cout << endl << endl;

        cout << "Please enter the radius of the second circle: ";
        cin >> circ2.radius;
        cout << "\nPlease enter the x-coordinate of the center: ";
        cin >> circ2.coordinate.x;
        cout << "\nPlease enter the y-coordinate of the center: ";
        cin >> circ2.coordinate.y;

        circ2.area = PI * pow(circ2.radius,2.0);
        circ2.circumference = 2 * PI * circ2.radius;

        cout << endl << endl;
        cout << setprecision(2) <<  fixed  <<  showpoint;

        cout << "The area of the first circle is :";
        cout << circ1.area << endl;
        cout << "The circumference of the first circle is: ";
        cout << circ1.circumference << endl;
        cout << "Circle 1 is centered at ("<< circ1.coordinate.x << ","
             << circ1.coordinate.y << ")." << endl << endl;

        cout << "The area of the second circle is :";
        cout << circ2.area << endl;
        cout << "The circumference of the second circle is: ";
        cout << circ2.circumference << endl;
        cout << "Circle 2 is centered at (" << circ2.coordinate.x << ","
            << circ2.coordinate.y << ")." << endl << endl;
        return 0;
}
```

Note that the programs in this lesson so far have not been modularized. Everything was done within the main function. In practice, this is not good structured programming. It can lead to unreadable and overly repetitious code. To solve this problem, we need to be able to pass structures and structure members to functions.

Structures and Functions

Just as we can use other variables as function arguments, structure members may be used as function arguments. Consider the following structure declaration:

```
struct circle
{
        double centerX;                 // x coordinate of center
        double centerY;                 // y coordinate of center
        double radius;
        double area;
};
```

Suppose we also have the following function definition in the same program:

```
float computeArea(float r)
{
        return PI * r * r;          // PI must previously be declared as a constant double
}
```

Let `firstCircle` be a variable of the `circle` structure type. The following function call will pass `firstCircle.radius` into `r`. The return value will be stored in `firstCircle.area`:

```
firstCircle.area = computeArea(firstCircle.radius);
```

It is also possible to pass an entire structure variable into a function rather than an individual member.

```
struct course
{
        char discipline[5];
        int courseNumber;
        char courseTitle[20];
        short credits;
};

course pChem;
```

Suppose the following function definition uses a `course` structure variable as its parameter:

```
void displayInfo(course c)
{
        cout << c.discipline << endl;
        cout << c.courseNumber << endl;
        cout << c.courseTitle << endl;
        cout << c.credits << endl;
}
```

Then the following call passes the `pChem` variable into `c`:

```
displayInfo(pChem);
```

Lab 8: Structures

The purpose of this lab is to introduce the concept of structures and to allow the students to manipulate and work with an array of structures. The lab will also give practical examples of using structures as parameters.

The lab begins with a quiz on the material covered in this lesson.

Fill-in-the-Blank Questions

1. The name of a structure is called the _____.

2. An advantage of structures over arrays is that structures allow one to use items of _____data types, whereas arrays do not.

3. One structure inside of another structure is an example of a _____.

4. The variables declared inside the structure declaration are called the_____ of the structure.

5. When initializing structure members, if one structure member is left uninitialized, then all the structure members that follow must be_____.

.

6. Another name for a user defined data type is a(n) _____.

7. Once an array of structures has been defined, you can access an array element using its _____.

8. The _____allows the programmer to access structure members.

9. You may not initialize a structure member in the _____.

10. Like variables, structure members may be used as_____ arguments.

Problem 8.1

Bring in program `rect_struct.cpp` from the Lab 8 folder. The code is shown below.

```
#include <iostream>
#include <iomanip>
using namespace std;

// This program uses a structure to hold data about a rectangle
// PLACE YOUR NAME HERE

// FILL IN CODE TO define a structure named rectangle which has
// members length, width, area, and perimeter all of which are floats
```

```
int main()
{
        // Fill IN CODE to declare a rectangle variable named box

        cout << "Enter the length of a rectangle: ";

       // FILL IN CODE to read in the length member of box

       cout << "Enter the width of a rectangle: ";

       // FILL IN CODE to read in the width member of box

       cout << endl << endl;

       // FILL IN CODE to compute the area member of box
       // FILL IN CODE to compute the perimeter member of box

       cout << fixed << showpoint << setprecision(2);

        // FILL IN CODE to output the area with an appropriate message
       // FILL IN CODE to output the perimeter with an appropriate message

       return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.

Exercise 2: Add code to the program above so that the modified program will determine whether or not the rectangle entered by the user is a square.

Sample Run:

Enter the length of a rectangle: 7
Enter the width of a rectangle: 7
The area of the rectangle is 49.00
The perimeter of the rectangle is 28.00
The rectangle is a square.

Problem 8. 2 Initializing Structures

Bring in program init_struct.cpp from the Lab 8 folder. The code is shown below.

```
#include <iostream>
#include<iomanip>
```

```
using namespace std;
// This program demonstrates partially initialized strucure variables

// PLACE YOUR NAME HERE

struct taxPayer
{
        char name[25];
        long int socialSecNum;
        float taxRate;
        float income;
        float taxes;
};

int main()
{
        // Fill in code to initialize a structure variable named citizen1 so that
        // the first three members are initilized.  Assume the name is Tim
        // McGuiness,the social security number is 255871234, and the tax rate is .35

        // Fill in code to initialize a structure variable named citizen2 so that
        // the first three members are initialized.  Assume the name is John Kane,
        // the social security number is 278990582, and the tax rate is .29

        cout << fixed << showpoint << setprecision(2);

        // calculate taxes due for citizen1

        // Fill in code to prompt the user to enter this year's income for the citizen1
        // Fill in code to read in this income to the appropriate structure member

        // Fill in code to determine this year's taxes for citizen1


        cout << "Name: " << citizen1.name << endl;
        cout << "Social Security Number: " << citizen1.socialSecNum << endl;

        cout << "Taxes due for this year: $" << citizen1.taxes << endl << endl;

        // calculate taxes due for citizen2

        // FILL IN CODE to prompt the user to enter this year's income for citizen2
        // FILL IN CODE to read in this income to the appropriate structure member

        // FILL IN CODE to determine this year's taxes for citizen2
```

```
        cout << "Name: " << citizen2.name << endl;
        cout << "Social Security Number: " << citizen2.socialSecNum << endl;

        cout << "Taxes due for this year: $" << citizen2.taxes << endl << endl;

        return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.

Sample Run:

Please input the yearly income for Tim McGuiness: 30000
Name: Tim McGuiness
Social Security Number: 255871234
Taxes due for this year: $10500.00

Please input the yearly income for John Kane: 60000
Name: John Kane
Social Security Number: 278990582
Taxes due for this year: $17400.00

Problem 8.3 Arrays of Structures

Bring in program array_struct.cpp from the Lab 8 folder. The code is shown below.

```
#include <iostream>
#include <iomanip>

using namespace std;

// This program demonstrates how to use an array of structures
// PLACE YOUR NAME HERE

// FILL IN CODE to define a structure called taxPayer that has three
// members:  taxRate, income, and taxes -- each of type float


int main()
{
  // FILL IN CODE to declare an array named citizen which holds
  // 5 taxPayers structures

        cout << fixed << showpoint << setprecision(2);
```

```
        cout << "Please enter the annual income and tax rate for 5 tax payers: ";
        cout << endl << endl << endl;

        for(int count = 0;count < 5;count++)
        {
                cout << "Enter this year's income for tax payer " << (count + 1);
                cout << ": ";

                // FILL IN CODE to read in the income to the appropriate place

                cout << "Enter the tax rate for tax payer # " << (count + 1);
                cout << ": ";

                // FILL IN CODE to read in the tax rate to the appropriate place

                // FILL IN CODE to compute the taxes for the citizen and store it
                // in the appropriate place

                cout << endl;
        }

        cout << "Taxes due for this year: " << endl << endl;

        // FILL IN CODE for the first line of a loop that will output the
        // tax information
        {
                cout << "Tax Payer # " << (index + 1) << ": " << "$ "
                        << citizen[index].taxes << endl;
        }

        return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.

Exercise 2: In the code above we have the following:

```
cout << "Tax Payer # " << (index+1) << ": " << "$ "
<< citizen[index].taxes << endl;
```

Why do you think we need `(index+1)` in the first line but index in the second?

Sample Run:

Enter this year's income for tax payer 1: 45000
Enter the tax rate for tax payer # 1: .19

Enter this year's income for tax payer 2: 60000
Enter the tax rate for tax payer # 2: .23
Enter this year's income for tax payer 3: 12000
Enter the tax rate for tax payer # 3: .01
Enter this year's income for tax payer 4: 104000
Enter the tax rate for tax payer # 4: .30
Enter this year's income for tax payer 5: 50000
Enter the tax rate for tax payer # 5: .22

Tax Payer # 1: $ 8550.00
Tax Payer # 2: $ 13800.00
Tax Payer # 3: $ 120.00
Tax Payer # 4: $ 31200.00
Tax Payer # 5: $ 11000.00

Problem 8.4 Nested Structures
Bring in program nestedRect_struct.cpp from the Lab 8 folder. The code is shown below.

```
#include <iostream>
#include <iomanip>

using namespace std;

// This program uses a structure to hold data about a rectangle
// It calculates the area and perimeter of a box

// PLACE YOUR NAME HERE

// Fill in code to define a structure named dimensions that
// contains 2 float members, length and width

// Fill in code to define a structure named rectangle that contains
// 3 members, area, perimeter, and sizes.  area and perimeter should be
// floats, whereas sizes should be a dimensions structure variable

int main()
{
  // Fill in code to declare a rectangle structure variable named box.

  cout << "Enter the length of a rectangle: ";

  // Fill in code to read in the length to the appropriate location

  cout << "Enter the width of a rectangle: ";

  // Fill in code to read in the width to the appropriate location
```

```
  cout << endl << endl;
  // Fill in code to compute the area and store it in the appropriate  location
  // Fill in code to compute the perimeter and store it in the  appropriate location

  cout << fixed << showpoint << setprecision(2);
  cout << "The area of the rectangle is " << box.attributes.area << endl;
  cout << "The perimeter of the recangle is " <<  box.attributes.perimeter
      << endl;

  return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.
Exercise 2: Modify the program above by adding a third structure named results which has two members area and perimeter**.** Adjust the rectangle structure so that both of its members are structure variables.
Exercise 3: Modify the program above by adding functions that compute the area and perimeter. The structure variables should be passed as arguments to the functions.
Sample Run:
Enter the length of a rectangle: 9
Enter the width of a rectangle: 6
The area of the rectangle is 54.00
The perimeter of the rectangle is 30.00

Student Generated Code Assignments

Student Generated Code Problem 8.5

Re-write Example 7.2 so that it works for an array of structures. Write the program so that it compares 6 circles. You will need to come up with a new way of determining which circle's center is closest to the origin.

Student Generated Code Problem 8.6

Write a program that uses a structure to store the following information for a particular month at the local airport:

Total number of planes that landed
Total number of planes that departed
Greatest number of planes that landed in a given day that month
Least number of planes that landed in a given day that month

The program should have an array of twelve structures to hold travel information for the entire year. The program should prompt the user to enter data for each month. Once all data is entered, the program should calculate and output the average monthly number of landing planes, the average monthly number of departing planes, the total number of landing and departing planes for the year,

and the greatest and least number of planes that landed on any one day (and which month it occurred in).

Lesson 8: Summary

An **abstract data type (ADT**) is a user defined data type.

A **structure** groups data items that can be of different data types

Example of a structure ADT;

```
struct course
{
        char discipline[5];
        int courseNumber;
        char  coursTitle[21];
        short credits;
};
```

The **tag** is the name of the structure, course in this case. The various variable definitions inside the braches are the **members** of the structure.

We can then create instances of the ADT.

```
course pChem;
course colonialHist;
```

The **dot operator** allows us to acess structure members.

```
colonialHist.credits = 3;
```

We can create **arrays of structures**

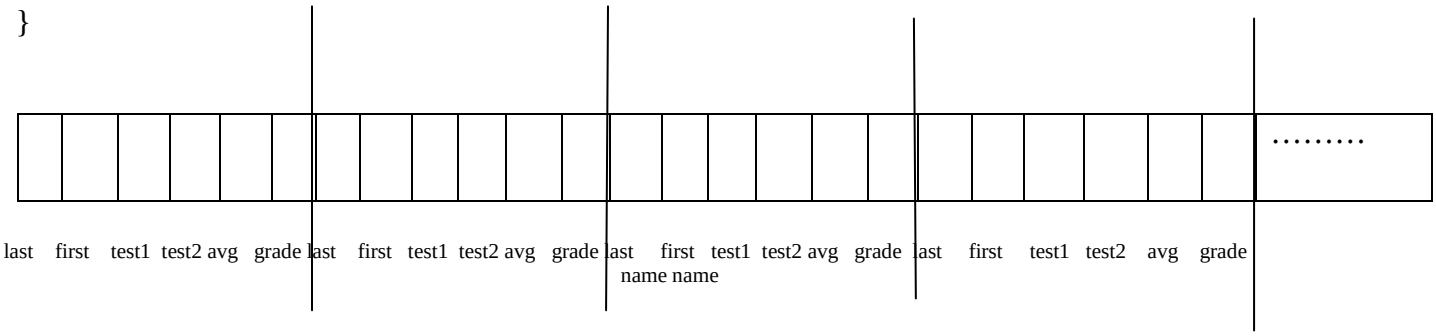
```
const MAX_STUDENTS = 200;

struct Student_data_type
{
  string      lastname;
  string      firstname;
  int         test1;
  int         test2;
  float       average;
  char        grade;
}  ;
```

```
typedef   Student_data_type   Student_array_type[MAX_STUDENT];

int main()
{
Student_array_type    student;
int pos = 0;

indata >> student[pos].lastname;
while(indata)
{
        indata >> student[pos].firstname >> student[pos].test1 >>  student[pos].test2;

        student[pos].average = (student[pos].test1 + student[pos].test2) / 2;
        // determine letter grade
              // instructions go here that will determine the letter grade

        //output information
        // instructions go here that will output the information
        pos++;
        indata >> student[pos].lastname
}
```

Questions What are the following data types?

student[pos].test 1 integer
student[pos] Student_data_type
student Student_array_type

}



Lesson 9: Classes and Objects

Review of Classes

You are already familiar with the concept of classes and objects from your study in Java. In fact, unlike C++, every Java program is defined within a class. In this lesson we review the class and object concept and apply it to the C++ language.

Up until now, we have been using the procedural programming method for writing all our programs. A procedural program has data stored in a collection of variables (or structures) and has a set of functions that perform certain operations. The functions and data are treated as separate entities. Although operational, this method has some serious drawbacks when applied to very large real-world situations. Even though this process is modularized (broken into several functions), in a large complex program the number of functions can become overwhelming and difficult to modify or extend. This can create a level of complexity that is difficult to understand.

Object-Oriented Programming (OOP) mimics real world applications by introducing **classes** which act as a prototype for **objects**. Objects are similar to nouns that can simulate persons, places or things that exist in the real world. OOP enhances code reuse ability (use of existing code or classes) so time is not used on "reinventing the wheel."

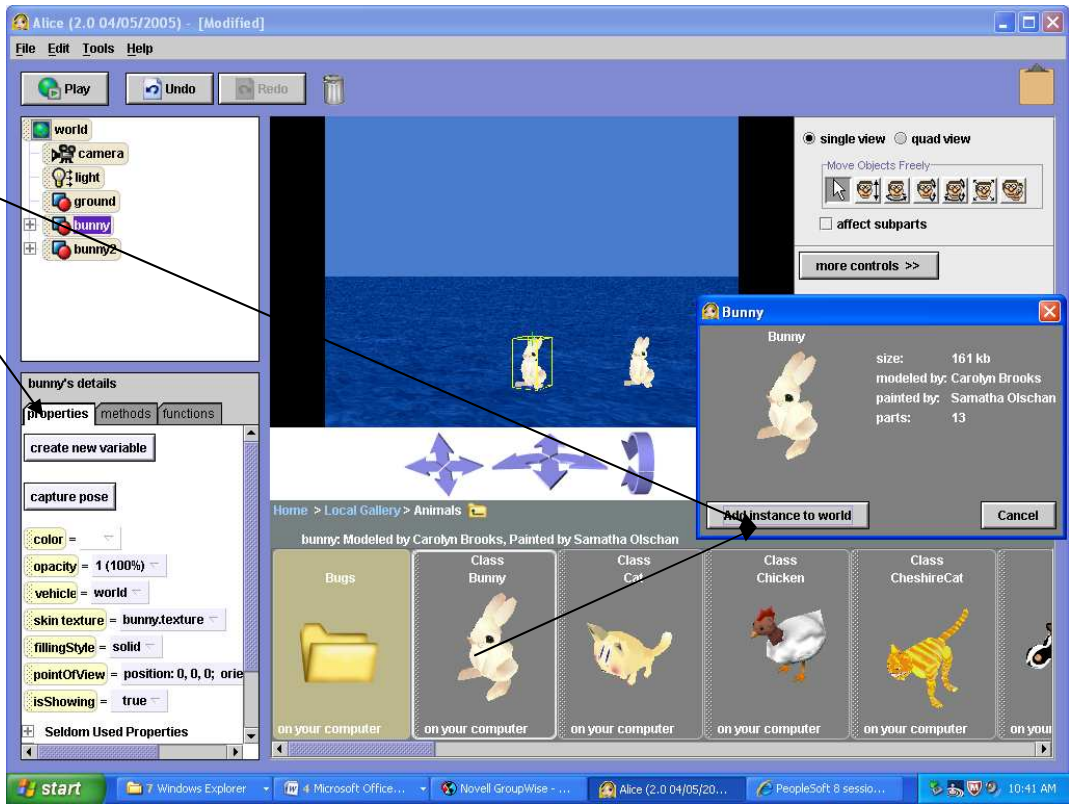
Classes and objects are often confused with one another; however, there is a subtle but important difference best explained by the following example. A plaster of Paris mold consists of the design of a particular figurine. When the plaster is poured into the mold and hardened, we have the creation of the figurine itself. A class is analogous to the mold, for it holds the definition of an object. The object is analogous to the figurine, for it is an **instance** of the class. Classes and structures are very similar in their construction. One difference is that classes may encapsulate functions with data declarations.

A **class** is a prototype (template) for a set of objects. An object can be described as a single instance of a class in much the same way that a variable is an instance of a particular data type. Just as several figurines can be made from one mold, many objects can be created from the same class. A class consists of a name (its identity), its member data which describes what it is and its member functions which describe what it does. Member data are analogous to nouns since they act as entities. Member functions are analogous to verbs in that they describe actions. A class is an **abstract data type (ADT)** which is a user defined data type that combines a collection of variables and operations. For example, a rectangle, in order to be defined, must have a length and width. In practical terms we describe these as its member data (length, width). We also describe a set of member functions that gives and returns valued to and from the ember data as wellas perform certain actions such as finding the rectangle's perimeter and area. Since many object can be created from the same class, each object must have its own set of member data.

You will recall these concepts from your study of Alice. You created an object by going to a list of classes (which were stored by category) and createing an instance of that class (the creation of an object). More than one object could be created from the same class. Each class was associated with procedures called methods that would have the object performing some action. Each object "inherited" or had acess to these methods. Each object also had certain properties or characteristics

that initially were the same for each object created from the same class but which could be altered by the programmer.

For example we created two bunny obects: instances from the class bunny. They each had certain properties such as color, opacity etc. that initially were the same; however we could go and change those properties. Each of the bunny objects however had access to the same methods (programs that performed some action) and functions(program sthat returned some value relating to the bunny object.)



As noted earlier, a class is similar to a structure except that classes encapsulate (contain) functions (methods) as well as data (properties). Functions and data items are usually designated as either **private** or **public** which indicates what can acess them. Data and functions that are defined as `public` can be directly accessed by code outside the class, while functions (methods) and data defined as `private` can be accessed only by functions belonging to the class. Usually, classes make data members `private` and require outside access to them through member functions that are difined as `public`. Member functions are thus usually defined as `public` and member data as `private`. The following example shows how a rectangle class can be defined in C++:

Example 9.1

```
#include <iostream>
using namespace std;

// Class declaration (header file)

class Rectangle     // Rectangle is the name of the class (its identity).
{

  public:
        // The following are labeled as public.  Usually member functions are defined public
        // and are used to describe what the class can do.
```

```
        void setLength(float side_l);
        // This member function receives the length of the Rectangle object that calls it
        // and places that value in the member data called length

        void setWidth (float side_w);
        // This member function receives the width of the Rectangle object that calls it
        // and places that value in the member data called width

        float getLength();
        // This member function returns the length of the Rectangle object that calls it

        float getWidth();
        // This member function returns the width of the Rectangle object that calls it

        double findArea();
        // This member function finds the area of the Rectangle object that calls it

        double findPerimeter();
        // This member function finds the perimeter of the Rectangle object that calls it

private:

        // The following are labeled as private.  Member data are usually declared private so they
        // can ONLY by accessed by functions that belong to the class.
        // Member data describe the attributes of the class

        float length;
        float width;
};
```

This example has six member functions. It has two member functions for each private member data: `setLength` and `getLength` for the member data `length` and `setWidth` and `getWidth` for the member data width. It is often the case that a class will have both a **set** and a **get** member function for each of its private data members. A set member function received a value from the calling object and places that value into the corresponding private member data. A get member function returns the value of the corresponding private member data to the object that calls it. In addition to set and get member functions, classes usually have other member functions that perform certain actions such as finding area and perimeter in the `Rectangle` class.

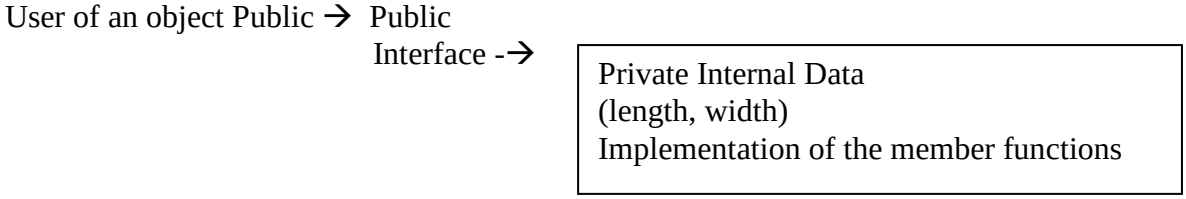
Client and Implementation Files

It is not necessary for someone to understand how a television remote control works in order to use the remote to change the stations or the volume. The user of the remote could be called a **client** that only knows how to use the remote to accomplish a certain task. The details of how the remote control performs the task are not necessary for the user to use the remote. Likewise, an automobile is a

complex mechanical machine with a simple interface that allows users without any (or very little) mechanical knowledge to start, drive and use it for a variety of functions. Drivers do not need to know what goes on under the hood. In the same way, a program that uses `Rectangle` does not need to know the details of how its member functions perform their operations. The use of an object (an instance of a class) is thus separated into two parts: the **interface** (client file) which calls the functions and the **implementation** which contains the details of how the functions accomplish their task.

An object not only combines data and functions but also restricts other parts of the program from accessing member data and the inner workings of member functions. Having programs or users access only certain parts of an object is called **data hiding**. The fact that the internal data and inner workings can be hidden from users makes the object more accessible to a greater number of programs.

Just like an automobile or a remote control, a piece of commercial software is usually a complex entity developed by many individuals. OOP (Object-Oriented Programming) allows programmers to create objects with hidden complex logic that have simple **interfaces** which are easily understood and used. This allows more sophisticated programs to be developed. Interfacing is a major concern for software developers.



Types of Objects

Objects are either general purpose or application-specific. General purpose objects are designed to create a specific data type such as currency or date. They are also designed to perform common tasks such as input verification and graphical output. Application-specific objects are created as a specific limited operation for some organization or task. A student class, for example, may be created for an educational institution.

Implementations of Classes in C++

The class declaration is usually placed in the global section of a program or in a special file (called a **header** file). As noted earlier, the class delaration acts very much like a prototype or data type for an object. An object is defined much like a variable except that it uses the class name as the data type. This definition creates an **instance** (actual occurrence) of the class. Implementation of the member functions of a class are given either after the `main` function of the program or in a separate file called the **implementation** file. Use of the object is usually in the `main` function, other specialized functions, or in a separate program file called the **client** file.

Creation and Use of Objects

`Rectangle`, previously described, is a class (prototype) and not an object (an actual instance of the class). Objects are defined in the client file, `main`, or other functions just as variables are defined:

```
Rectangle box1, box2;
```

box1 and box2 are objects of class Rectangle
.
box1 has it's own length and width that are possible different from the length and width of box2.

To access a member function (method) of an object, we use the dot operator, ujust as we do to access data members of structures. The name of the object is given first, followed by the dot operator and then the name of the member function.

Example 9.2 shows a complete main function (or client file) that defines and uses objects which call member functions.

Example 9.2

```
int main()
{
        Rectangle box1;             // box1 is defined as an object of Rectangle class
        Rectangle box2;             // box 2 is defined as another Rectangle class object

        box1.setLength(20);         // This instruction has the object box1 calling the setLength
                                    // member function which sets the member data length associated
                                    // with box1 to the value of 20.

        box1.setWidth(5);
        box2.setLength(9.5);        // This instruction has the object box2 calling the setLength
                                    // member function which sets the member data length associated
                                    // with box2 to the value of 9.5
        box2.setWidth(8.5)

        cout << "The length of box1 is " << box1.getLength() << endl;
        cout << "The width of box1 is " << box1.getWidth() << endl;
        cout << "The area of box1 is " << box1.getArea() << endl;
        cout << "The perimeter of box1 is " << box1.findPerimeter() << endl;

        cout << "The length of box2 is " << box2.getLength() << endl;
        cout << "The width of box2 is " << box2.getWidth() << endl;
        cout << "The area of box2 is " << box2.getArea() << endl;
        cout << "The perimeter of box2 is " << box2.findPerimeter() << endl;

        return 0;
}
```

Since findArea and findPerimeter must have the length and width before they can do the calculation,
an object must call setLength and setWidth first. The user must remember to initialize both length

and `width` by calling both set functions. It is not good programming practice to assume that a user will do the necessary initialization. Constructors (discussed later) solve this problem.

Implementation of Member Functions

As previously noted, the implementation of the member function can be hidden from the users (clients) of the objects. However, they must be implemented by someone, somewhere. The following shows the implementation of the `Rectangle` member functions
member functions.

Example 9.3

```
//*********************************************************************************
//                                    setLength
//
//      task:           This member function of the class Rectange receives
//                      the length of the Rectangle object that calls it and
//                      places that value in the member data called length.
//
//      data in:        the length of the rectangle
//      data out:       none
//*********************************************************************************

void Rectangle::setLength(float side_1)
{
        length = side_1;
}

//*********************************************************************************
//                                    setWidth
//
//      task:           This member function of the class Rectange receives
//                      the width of the Rectangle object that calls it and
//                      places that value in the member data called length.
//
//      data in:        thewidth of the rectangle
//      data out:       none
//*********************************************************************************

void Rectangle::setWidth(float side_w)
{
        width = side_w;
}


//*********************************************************************************
//                                    getLength
//
```

```
//      task:           This member function of the class Rectange returns
//                      the length of the Rectangle object that calls it.
//
//      data in:        none
//      data out:       length
//*************************************************************************

float Rectangle::getLength()
{
        return length;
}

//*************************************************************************
//                              getWidth
//
//      task:           This member function of the class Rectange returns
//                      the width of the Rectangle object that calls it.
//
//      data in:        none
//      data out:       width
//*************************************************************************

float Rectangle::getWidth()
{
        return width;
}

//*************************************************************************
//                              findArea
//
//      task:           This member function of the class Rectange calculates
//                      the area of the Rectangle object that calls it.
//
//      data in:        none (uses the values of member data length & width)
//      data returned  area
//*************************************************************************

double Rectangle::findArea()
{
        return length * width;
}

//*************************************************************************
//                              findPerimeter
//
//      task:           This member function of the class Rectange calculates
```

```
//                          the perimeter of the Rectangle object that calls it.
//
//      data in:            none (uses the values of member data length & width)
//      data returned  perimeter
//*************************************************************************

double Rectangle::findPerimeter()
{
        return ( ( 2* length) + ( 2* width) );
}
```

Notice that in the heading of each member function, the name of the function is preceded by the name of the class to which it is a member followed by a double colon. In the above example each name is preceded by **`Rectangle::`**. This is necessary to indicate in which class the function is a member. There can be more than one function with the same name associated with different classes. The :: symbol is called the **scope operator**. It acts as an indicator of the class association.

Usually classes are declared in a header file while member functions are stored in an implementation file and objects are declared and used in a client file. These files are often bound together in a project. Various development environments have different means of creating and storing related files in a project. All could be located in three different sections of the same file.

Complete Program

The following code shows the class declaration, member functions (methods), implementations and use (client) of the `Rectangle` class:

Example 9.4

```
#include <iostream >
using namespace std;

//
// Class declaration (header file)

class Rectangle          // Rectangle is the name of the class (its identity).

{
public:
        // The member functions are labeled as public

void setLength (float side_1);
// This member function receives the length of the Rectangle object that calls it and places that
// value in the member data called length.

void setWidth(float side_w);
```

```
// This member function receives the width of the Rectangle object that calls it and places that
// value in the member data called width.

void getLength ();
// This member function returns the length of the Rectangle object that calls it.

void getWidth ();
// This member function returns the width of the Rectangle object that calls it.

double findArea ();
// This member function finds the area of the Rectangle object that calls it.

double findPerimeter();
// This member function finds the perimeter of the Rectangle object that calls it.
void CircleStats();
// This member function outputs the radius and center of the circle object
// that calls it.

private:

//  The following are labeled as private.
// Member data are usually defined private so they can ONLY  be accessed by functions that
// belong to the class.
// Member data describe what the attribute of the class

float length;
float width;
float center_y;
};          // Notice the braces and the semicolon at the end.

//_____________________________________________________________________________
// Client file

int main()
{
        Rectangle box1;             // box1 is defined as an object of Rectangle class
        Rectangle box2;             // box 2 is defined as another Rectangle class object

        box1.setLength(20);         // This instruction has the object box1 calling the setLength
                                    // member function which sets the member data length associated
                                    // with box1 to the value of 20.

        box1.setWidth(5);
        box2.setLength(9.5);        // This instruction has the object box2 calling the setLength
                                    // member function which sets the member data length associated
                                    // with box2 to the value of 9.5
```

```
        box2.setWidth(8.5)

        cout << "The length of box1 is " << box1.getLength() << endl;
        cout << "The width of box1 is " << box1.getWidth() << endl;
        cout << "The area of box1 is " << box1.getArea() << endl;
        cout << "The perimeter of box1 is " << box1.findPerimeter() << endl;

        cout << "The length of box2 is " << box2.getLength() << endl;
        cout << "The width of box2 is " << box2.getWidth() << endl;
        cout << "The area of box2 is " << box2.getArea() << endl;
        cout << "The perimeter of box2 is " << box2.findPerimeter() << endl;

        return 0;
}
```

//*******************
// Implementation file

```
//*************************************************************************
//                                  setLength
//
//      task:           This member function of the class Rectange receives
//                      the length of the Rectangle object that calls it and
//                      places that value in the member data called length.
//
//      data in:        the length of the rectangle
//      data out:       none
//*************************************************************************

void Rectangle::setLength(float side_1)
{
        length = side_1;
}

//*************************************************************************
//                                  setWidth
//
//      task:           This member function of the class Rectange receives
//                      the width of the Rectangle object that calls it and
//                      places that value in the member data called length.
//
//      data in:        thewidth of the rectangle
//      data out:       none
//*************************************************************************
```

```
void Rectangle::setWidth(float side_w)
{
        width = side_w;
}


//*************************************************************************
//                                  getLength
//
//      task:           This member function of the class Rectange returns
//                      the length of the Rectangle object that calls it.
//
//      data in:        none
//      data out:       length
//*************************************************************************

float Rectangle::getLength()
{
        return length;
}

//*************************************************************************
//                                  getWidth
//
//      task:           This member function of the class Rectange returns
//                      the width of the Rectangle object that calls it.
//
//      data in:        none
//      data out:       width
//*************************************************************************

float Rectangle::getWidth()
{
        return width;
}

//*************************************************************************
//                                  findArea
//
//      task:           This member function of the class Rectange calculates
//                      the area of the Rectangle object that calls it.
//
//      data in:        none (uses the values of member data length & width)
//      data returned  area
//*************************************************************************

double Rectangle::findArea()
```

```
{
        return length * width;
}

//*************************************************************************
//                                  findPerimeter
//
//      task:           This member function of the class Rectange calculates
//                      the perimeter of the Rectangle object that calls it.
//
//      data in:        none (uses the values of member data length & width)
//      data returned  perimeter
//*************************************************************************

double Rectangle::findPerimeter()
{
        return ( ( 2* length) + ( 2* width) );
}
```

Inline Member Functions

Sometimes the implementation of member functions is so simple that they can be defined inside a class declaration. Such functions are called inline member functions. In the `Rectangle` class, `findArea` and `findPerimeter` are so simple that they can be defined in the class declaration as follows:

```
double findArea()  {  return length * width; }
double findPerimeter()  {  return  2 *length + 2* width; }
```

Introdution to Constructors

As noted earlier, the methods (member functions) `findArea` and `findPerimeter` must have the `length` and `width` before they can do any calculation. The user must remember to initialize both length and width by calling both of these functions. What happens if the user forgets? Suppose we call `findArea` without first calling both `setLength` and `setWidth`. The function will try to find the area of a rectangle that has no length or width. Thus, the creator of a class should never rely on the user to initialize essential data.

C++ provies a mechanism, called a **constructor,** to guarantee the initialization of an object. A constructor is a member function that is implicitily invoked whener a class instance is created (whenever an object is defined). A constructor is unique from other member functions in two ways:

1. It has the same name as the class itself.
2. It does not have a data type (or the word void) in front of it. The only purpose of the constructor is to initialize an object's member data.

The following shows the rectangle class using two constructors that set the values of length and width.

Example 9.5

```
class Rectangle
{
public:
        Rectangle (float side_l, float side_w);
                // Constructor allowing a user to input the length and width
        Rectangle();
                // Constructor using default values for both length and width

        void setLength(float side_l);
        void setWidth(float side_w);
        float getLength();
        float getWidth();
        double findArea();
        double findPerimeter();
private:
        float length;
        float width;
};
```

This class includes two constructors, differentiated by their parameter lists. Recall that two or more functions can have the same name as long as their parameters differ in quantity or data type. The parameter-less constructor (the second constructor in the above example) is the **default constructor**. Like all member functions, constructors are defined in the implementation file (or function definition section of a program). The reason for a default constructor is explained in the next section.

Constructor Defintions

The function definitions of the two constructors for the Rectangle class are as follows:

```
Rectangle::Rectangle(float side_l, float side_w)
{
        length = side_l;
        width = side_w;
}

Rectangle::Rectangle()
{
        length = 1;
        width = 1;
}
```

The first constructor allows the user to input a value for both `length` and `width` at the same time that the object is defined. The second constructor (the default constructor) sets both `length` and `width` to 1 whenever the object is defined. Actually they could be set to anything that the creator of the class wants to use as a default for an object of the class that is not initialized by the user. With the use of these constructors, every object of class `Rectangle` will have a value for both `length` and `width`. We still keep the two member functions `setLength` and `setWidth` to allow the user to change the value of length and width. We could create a third constructor that has just one parameter which gives the value of `length` and uses the default value for `width`. If we create this third constructor, however we can not create a fourth constructor that gives the value of `width` and uses the default value for `length`. Why? We would have two member functions with the same name and an identical parameter list in both data type and number.

Invoking a Constructor

Although a constructor is a member function, it is never invoked (called) using the dot notation. It is invoked when an object is defined.

Example:
```
Rectangle box1(12,6);
Rectangle box2;
```

In this example, `box1` is an object of `Rectangle` class that has `length` set to 12 and `width` set to 6. Since it has twoparameters, box1 activates the constructor that has two parameters. The object `box2` is defined with both `length` and `width` set to 1. Since `box2` has no parameters, it activates the default constructor.

Destructors

A **destructor** is a member function that is automatically called to destroy an object. Just like constructors, a destructor has the same name as the class: however, it is preceded by a tilde (~). Destructors are used to free up memory when the object is no longer needed. The destructor is automatically called when an object of the class goes out of scope. This occurs when the function (such as main), where the object is defined, ends. The following example show how constructors and destructors operate.

Example 9.6

```
#include <iostream>
using namespace std;

class Demo
{
public:
        Demo();             // Default constructor
        ~Demo();            // Destructor
};
```

```
int main()

{
        Demo demoObj;       // demoObj is defined and invokes the default constructor that
                            // prints the message "The constructor has been invoked"

        cout  <<  "The program is now running"  << endl;
        return 0;
}

//  Now that the main program is over, the object demoObj is no longer active.  The destructor i
// invoked and the message "The destructor has been invoked" is printed.

//*************************************************************************
//                              The default Constructor Demo
//
//      Notice that constructors do not have to set member data
//      This constructor prints a message that the constructor has been invoked
//*************************************************************************

Demo::Demo()
{
        cout  << "The constructor has been invoked"  << endl;
}

//*************************************************************************
//                              The Destructor Demo
//
//      Notice that constructors do not have to print anything but this destructor
//      prints the message "The destructor has been invoked."  The primary prupose of
//      destructors is to free memory space once an object is no longer needed
//*************************************************************************

Demo::~Demo()
{
        cout  << "The destructor has been invoked"  << endl;
}
```

What order do you think the three `cout` statements will be executed? Note that a class can have only one default constructor and one destructor.

Array of Objects

Arrays can also contain objects of a class. For example, we could have an array or Rectangle objects.

Example:

Rectangle box[4]; // box is defined as an array of Rectangle objects

This statement makes an array of 4 elements, each consisting of an object of the Rectangle class.

Since this class has a default constructor, the default values are assigned to each element (object of the array. The length and width for each of the objects in the box array are equal to 1 since these are the default values assigned by the default constructor.

The following example demonstrates the use of an array of objects.

Example 9.7

```
#include <iostream>
using namespace std;

class Rectangle
{
public:

        // Constructor allowing a user to input the length and width
        Rectangle(float side_l, float side_w);
        Rectangle();                    // Default constructor
        ~Rectangle();                   // Destructor

        void setLength(float side_l);
        void setWidth(float side_w);
        float getLength();
        float getWidth();
        double findArea();
        double findPerimeter();

private:

        float length;
        float width;
};

const int NUMBEROFOBJECTS = 4;

int main()
{
        Rectangle box(NUMBEROFOBJECTS);  // Box is defined as an arry of Rectangle objects

        for (int pos = 0; pos < NUMBEROFOBJECTS; pos++)
```

```
        {
                cout   << "Information for box number " << pos + 1 << endl << endl;
                cout  << "The length of the box is "  << box[pos].getLength << endl;
                cout  << "The width of the box is "  << box[pos].getWidth << endl;
                cout  << "The area of the box is "  << box[pos].findArea << endl;
                cout  << "The perimeter of the box is "  << box[pos].findPerimeter << endl;
        }

        return 0;
}

void Rectangle:: setLength(float side_l)
{
        length = side_l;
}

void Rectangle:: setWidth(float side_w)
{
        width = side_w;
}

float Rectangle:: getLength()
{
        return length;
}

float Rectangle:: getWidth()
{
        return width;
}

double Rectangle:: findArea()
{
        return length * width;
}

double Rectangle:: findPerimeter()
{
        return ( ( 2 *length) + (2 * width)  );
}

Rectangle::Rectangle(float side_l, float side_w)
{
        length = side_l;
        width = side_w;
}
```

```
Rectangle::Rectangle()
{
        length = 1;
        width = 1;
}

Rectangle::~Rectangle()
{

}
```

Lab 9: Classes and Objects

The purpose of this lab is to introduce the concept of classes, constructors, destructors and the use of an array of objects

The lab begins with a quiz on the material covered in this lesson.

Fill-in-the-Blank Questions

1. A(n)_____ is used in C++ to guarantee the initialization of a class instance.

2. A constructor has the _____name as the class itself.

3. Member functions are sometimes called _____in other Object oriented languages.

4. A(n) _____ is a member function that is automatically called to destroy an object.

5. To access a particular member function, the code must list the object name and the name of the function separated from each other by a _____.

6. A _____constructor has no parameters.

7. A(n) _____precedes the destructor name in the declaration.

8. A(n) _____member function has its implementation given in the class declaration.

9. In an array of objects, if the default constructor is invoked, then it is applied to _____object in the array.

10. A constructor is a member function that is _____invoked whenever a class instance is created.

Problem 9.1 Squares as a Class

Retrieve program `square.cpp` from the Lab 9 folder. The code is as follows:

```
#include <iostream>
using namespace std;

// FILL IN THE CODE TO DECLARE A CLASS CALLED Square.  TO DO THIS SEE
// THE IMPLEMENTATION SECTION.

int main()
{
   Square  box;         // box is defined as an object of the Square class
```

```
   float   size;        // size contains the length of a side of the square

  // FILL IN THE CLIENT CODE THAT WILL ASK THE USER FOR THE LENGTH OF THE
  // SIDE OF THE SQUARE.  (This is stored in size)

  // FILL IN THE CODE THAT CALLS SetSide.

  // FILL IN THE CODE THAT WILL RETURN THE AREA FROM A CALL TO A FUNCTION
  // AND PRINT OUT THE AREA TO THE SCREEN

  // FILL IN THE CODE THAT WILL RETURN THE PERIMETER FROM A CALL TO A
  // FUNCTION AND PRINT OUT THAT VALUE TO THE SCREEN

        return 0;
}

//_____________________________________________________________________
//Implementation section     Member function implementation

//*************************************************
//               setSide
//
// task:    This procedure takes the length of a side and
//          places it in the appropriate member data
// data in: length of a side
//*************************************************

void Square::setSide(float length)

{
  side = length;
}
//*************************************************
//               findArea
//
// task:    This finds the area of a square
// data in: none (uses value of data member side)
// data returned:  area of square
//*************************************************

float Square::findArea()
{
  return side * side;
}
//*************************************************
//               findPerimeter
//
```

```
// task:    This finds the perimeter of a square
// data in: none (uses value of data member side)
// data returned:  perimeter of square
//*****************************************************
float Square::findPerimeter()
{
  return 4 * side;
}
```

Exercise 1: This program asks you to fill in the class declaration and client code based on the implementation of the member functions. Fill in the code so that the following input and output will be generated:

Please input the length of the side of the square
8
The area of the square is 64
The perimeter of the square is 31

Exercise 2: Add two constructors and a destructor to the class and create the implementation of each. One constructor is the default constructor that sets the side to 1. The other constructor will allow the user to initialzie the side at the defintion of the object. The destructor does not have to do anything but reclaim memory space. Create an object called box1 that gives the value of 9 to the constructor at the defintion. Add output statements so that the following is printed in additon to what is printed in Exercise 1.

The area of box1 is 81
The perimeter of box1 is 36

Problem 9.2 Circles as a Class

Retrieve program `circles.cpp` from Lab 9 folder. The code is as follows

---

```
#include <iostream>
using namespace std;
//_____________________________________________________________________
// This program declares a class for a circle that will have
// member functions that set the center, find the area, find
// the circumference and display these attributes.
// The program as written does not allow the user to input data, but
// rather has the radii and center coordinates of the circles (spheres in the program)
// initialized  at declaration or set by a function.

//class declaration section   (header file)

class Circles
{
```

```
public:
   void setCenter(int x, int y);
   double findArea();
   double findCircumference();
   void printCircleStats();       // This outputs the radius and center of the circle.
   Circles (float r);             // Constructor
   Circles();                     // Default constructor
private:
   float  radius;
   int    center_x;
   int    center_y;
};


const double PI = 3.14;

//Client section

int main()
{
   Circles sphere(8);
   sphere.setCenter(9,10);
   sphere.printCircleStats();

   cout << "The area of the circle is " << sphere.findArea() << endl;
   cout << "The circumference of the circle is "
              << sphere.findCircumference() << endl;

   return 0;
}

//_____________________________________________________________________
//Implementation section     Member function implementation

Circles::Circles()
{
   radius = 1;
}
```

// Fill in the code to implement the non-default constructor

// Fill in the code to implement the findArea member function

// Fill in the code to implement the findCircumference member function

```
void Circles::printCircleStats()
// This procedure prints out the radius and center coordinates of the circle
// object that calls it.

{
  cout << "The radius of the circle is " << radius << endl;
  cout << "The center of the circle is (" << center_x
       << "," << center_y << ")" << endl;
}

void Circles::setCenter(int x, int y)
// This procedure will take the coordinates of the center of the circle from
// the user and place them in the appropriate member data.

{
  center_x = x;
  center_y = y;
}
```

Exercise 1: Alter the code so that setting the center of the circle is also done during the object defintion. This means that the constructors will also take care of this initialization. Make the default center at point (0,0) and keep the default radius as 1. Have `sphere` defined with initial values of 8 for the radius and (9,10) for the center. How does this affect existing functions and code in the `main` function?

The following output should be produced:

The radius of the circle is 8
The center of the circle is (9,10)
The area of the circle is 200.96
The circumference of the circle is 50.25

Exercise 2: There can be several constructors as long as they differ in number of parameters or data type. Alter the program so that the user can enter either just the radius, the radius and the center, or nothing at the time the object is defined. Whater the user does NOT include (radius or center) must be initialized somewhere. There is no `setRadius` fucntion and there will no longer be a `setCenter` function. You can continue to assume that the dfault radius is 1 and the default center is (0,0). Alter the client portion(`Main`) of the program by defining an object `sphere1`, giving just the radius of 2 and the default center, and `sphere2` by giving neither the radius nor the center (it uses all the default values). Be sure to print out the vital statistics for these new objects (area and circumference).

In addition to the output in Exercise 1, the following output should be included:

The radius of the circle is 2
The center of the circle is (0,0)
The area of the circle is 12.56

The circumference of the circle is 12.56

The radius of the circle is 1
The center of the circle is (0,0)
The area of the circle is 3.14
The circumference of the circle is 6.28

Exercise 3: Alter the program you generated in Exercise 2 so that the user will be allowed to enter either nothing, just the radius, just the center, or both the center and radius at the time the object is defined. Add to the client portion of the code an object called spher3 that, when defined, will have the center at (15,16) and the default radius. Be sure to print out this new object's vital statistics (area and circumference).

In addition to the output in Exercise 1 and 2, the following output should be printed:

The radius of the circle is 1
The center of the circle is (15,16)
The area of the circle is 3.14
The circumference of the circle is 6.28

Exercise 4: Add a destructor to the code. It should print the message **This concludes the Circles class** for each object that is destroyed. How many times is this printed? Why?

Problem 9.3 Arrays as Data Members of Classes
Retried program `floatarray.cpp` and `temperatures.txt` from the Lab 9 folder. The code is as follows:

```
// This program reads floating point data from a data file and places those
// values into the private data member called values (a floating point array)
// of the FloatList class. Those values are then printed to the screen.
// The input is done by a member function called GetList. The output
// is done by a member function called PrintList.  The amount of data read in
// is stored in the private data member called length.  The member function
// GetList is called first so that length can be initialized to zero.

#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

const int MAX_LENGTH = 50;        // MAX_LENGTH contains the maximum length of our list
class FloatList                   // Declares a class that contains an array of floating
                                  // point numbers
{
public:
        void getList(ifstream&);  // Member function that gets data from a file
```

```
        void printList() const;       // Member function that prints data from that
                                      // file to the screen.
        FloatList();                  // constructor that sets length to 0.
        ~FloatList();                 // destructor

private:
        int length;                           // Holds the number of elements in the array
        float values[MAX_LENGTH];             // The array of values

};

int main()
{
  ifstream tempData;   // Defines a data file

  // Fill in the code to define an object called list of the class FloatList

  cout << fixed << showpoint;
  cout << setprecision(2);

  tempData.open("temperatures.txt");


  // Fill in the code that calls the getList function.
  // Fill in the code that calls the printList function.

return 0;
}



 FloatList::FloatList()
 {
        // Fill in the code to complete this constructor that
        // sets the private data member length to 0
 }

// Fill in the entire code for the  getList function
// The getList function reads the data values from a data file into the values array of the class
// FloatList

// Fill in the entire code for the  printList function
// The printList function prints to the screen the data in
// the values array of the class FloatList

 // Fill in the code for the implementation of the destructor
```

This program has an array of floating point numbers as a private data member of a class. The data file contains floating point temperatures which are read by a member function of the class and stored in the array.

Exercise 1: Why does the member function printList have a const after its name but getList does not?

Exercise 2: Fill in the code so that the program reads in the data values from the temperature file and prints them to the screen with the following output:

78.90
87.40
60.80
70.40
75.60

Exercise 3: Add code (member function, call and fucntion implementation) to print the average of the numbers to the screen so that the output will look like the output from Exercise 2 plus the following:

The average temperature is 74.62

Problem 9.4 Arrays of Objects

Retrieve program `inventory.cpp` and `inventory.dat` from the Lab 9 folder. The code is as follows:

```
#include <iostream>
#include <fstream>
using namespace std;


// This program defines a class called Inventory that has itemnumber (which
// contains the id number of a product) and numofitem (which contains the
// quantity on hand of the corresponding product)as private data members.
// The program will read these values from a file and store them in an
// array of objects (of type Inventory).  It will then print these values
// to the screen.

// Example: Given the following data file:
//     986 8
//     432 24
// This program reads these values into an array of objects and prints the
// following:
//     Item number 986 has 8 items in stock
//     Item number 432 has 24 items in stock
```

```
const NUMOFPROD = 10;   // This holds the number of products a store sells

class Inventory
{
public:

   void getId(int item);            // This puts item in the private data member
                                    // itemnumber of the object that calls it.
   void getAmount(int num);    // This puts num in the private data member
                               // numofitem of the object that calls it.
   void display();             // This prints to the screen
                                // the value of itemnumber and numofitem of the
                                // object that calls it.
private:

   int  itemNumber;            // This is an id number of the product
   int  numOfItem;             // This is the number of items in stock

};


int main()
{

   ifstream infile;            // Input file to read values into array
   infile.open("Inventory.dat");

   // Fill in the code that declares an array of objects of class Inventory
   // called products. The array should be of size NUMOFPROD

   int pos;                 // loop counter
   int id;                  // variable holding the id number
   int total;               // variable holding the total for each id number

   // Fill in the code that will read inventory numbers and number of items
   // from a file into the array of objects. There should be calls to both
   // getId and getAmount member functions somewhere in this code.
   // Example: products[pos].getId(id); will be somewhere in this code

   // Fill in the code to print out the values (itemNumber and numOfItem) for
   // each object in the array products.
   // This should be done by calling the member function display within a loop

   return 0;

}
```

// Write the implementations for all the member functions of the class.

Exercise 1: Complete the program by giving the code explained in the commands in bold. The data file is as follows:

```
986  8
432  24
132  100
123  89
329  50
503  30
783  78
822  32
233  56
322  74
```

The output should be as follows:

Item number 987 has 8 items in stock
Item number 432 has 24 items in stock
…
etc.

Student Generated Code Assignments

Student Generated Code Problem 9.5

Exercise 1: Give a C++ class declaration called `SavingsAccount` with the following information:

Operations (Member Functions)

1. Open account (with an initial deposit). This is called to put initial values in `dollars` and `cents`.

2. Make a deposit. A function that will add value to `dollars` and `cents`

3. Make a withdrawal. A function that will subtract values from `dollars` and `cents`

4. Show current balance. A function that will print `dollars` and `cents`.

Data (Member Data)

1. `dollars`
2. `cents`

Give the implementation code for all the member functions.
NOTE: You must perform normalization on `cents`. This means that if `cents` is 100 or more, it must

increment `dollars` by the appropriate amount. Example: if `cents` is 234, then `dollars` must be increased by 2 and `cents` reduced to 34.

Write code that will create an object called `bank1`. The code will then initially place $200.50 in the account. The code will deposit $40.50 and then withdraw $100.98. It will print out the final value of `dollars` and `cents`.

The following output should be produced:

Dollars = 40 cents = 2.

Exercise 2: Change the program to allow the user to input the initial values, deposit and withdrawal.

Example:

Please input the initial dollars
402
Please input the initial cents
78
Would you like to make a deposit? Y or y for yes
y
Please input the dollars to be deposited
35
Please input the cents to be deposited
67
Would you like to make a deposit? Y or y for yes
y
Please input the dollars to be deposited
35
Please input the cents to be deposited
67
Would you like to make a deposit? Y or y for yes
n
Would you like to make a withdrawal Y or y for yes
y
Please input the dollars to be withdrawn
28
Please input the cents to be withdrawn
08
Would you like to make a withdrawal Y or y for yes
y
Please input the dollars to be withdrawn
75
Please input the cents to be withdrawn
78

Would you like to make a withdrawal Y or y for yes
n

Dollars = 370 Cents = 26

Exercise 3: Replace the initial member function by two constructors. One constructor is the default constructor that sets both `dollars` and `cents` to 0. The other constructor has 2 parameters that set `dollars` and `cents` to the indicated values.

Have the code generate two objets: `bank1` (which has its values set during definition by the users) and `bank2` that uses the default constructor. Have the code input deposits and withdrawals for both `bank1` and `bank2`.

Lesson 9: Summary

Object-Oriented Programming (OOP) mimics real world applications by introducing **classes** which act as a prototype for **objects**.

A **class** is a prototype (template) for a set of objects. An **object** can be described as a single instance of a class. A class is an **abstract data type (ADT)** which is a user defined data type that combines a collection of variables and operations.

Functions and data items are usually designated as either **private** or **public** which indicates what can acess them. Data and functions that are defined as `public` can be directly accessed by code outside the class, while functions (methods) and data defined as `private` can be accessed only by functions belonging to the class. Usually, classes make data members `private` and require outside access to them through member functions that are difined as `public`. Member functions are thus usually defined as `public` and member data as `private`.

The class declaration is usually placed in the global section of a program or in a special file (called a **header** file). Implementation of the member functions of a class are given either after the `main` function of the program or in a separate file called the **implementation** file. Use of the object is usually in the `main` function, other specialized functions, or in a separate program file called the **client** file.

Class declaration example:

```
Class Rectangle
{
public:
        void setLength(float side_l);
        void setWidth(float side_w);
        float getLength();
        float getWidth();
        double finArea();
private:
        float length;
        float width;
};
```

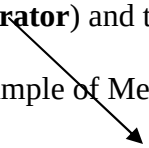
Objects are defined in the client file, main or other functions just as variables are defined.

```
Rectangle  box1;
```

Class member functions are used by listing the object followed by the dot operator followed by the name of the function.

```
box1.setLength(20);
```

Implementation of Member function must include the name of the class followed by :: (the **scope operator**) and the name of the function itself.



Example of Member function implementation

```
void Rectangle::setLength(float side_l)
{
        length = side_l;
}
```

C++ provies a mechanism, called a **constructor,** to guarantee the initialization of an object. A constructor is a member function that is implicitily invoked whener a class instance is created (whenever an object is defined). A constructor is unique from other member functions in two ways:

1. It has the same name as the class itself.
2. It does not have a data type (or the word void) in front of it. The only purpose of the constructor is to initialize an object's member data.

A destructor is a member function that is automatically called to destroy an object. A destructor has the same name as the calss itself except the name is preceded by a tilde (~).

Arrays can contain objects of a class.

Lesson 10: Pointers

Reference Data Types

The basic idea of pointers is not unfamiliar to us. In Java you learned that there were two data types: primitive and reference. A variable that contains a reference type (which is really a class) stores an address to some location rather than what we normally think of as data. We have also seen this concept in arrays where the "variable" name of an array is really an address that points to the first element of the array.

C++ extends this concept into what is called **pointers**.

Pointer Variables

A distinction must always be made between a memory location's address and the data stored at that location. A street address like 119 Main St. is a location that is different than a description of what is at that location: the little red house of the Smith family. So far we have been concerned only with data stored in a variable, rather than with its address (where in main memory the variable is located). In this lesson we look aat addresses ofvariables and at special variables called **pointers,** which hold these addresses. The address of a variable is given by preceding the variable name with the C++ address operator (**&**):

```
cout  <<  &sum;   // This outputs the address of the variable sum and not the contents of the memory  location.
```

The (`&`) operator in front of a variable `sum` indicates that the address itself, not the data stored in that location, is the value used. On most systems the above address will print as a hexadecimal value representing the physical location of the variable. Before this lesson, where have you used this (&) symbol in C++ programming? You may recall that it was used in the prototype and the function heading of a function for parameters being passed by reference. This connection will be explored in the next section.

To define a variable to be a pointer, we precede it with an asterisk (*):

```
int *ptr;
```

The asterisk in front of the variable `ptr` indicates that `ptr` holds the address of a memory location. The `int` indicates that the memory location that `ptr` points to holds integer values. `ptr` is NOT an integer data type but rather a pointer thatl holds the address of a location where an integer value is stored. This distinction is most important!

The following example illustrates the difference

```
int sum;                    // sum holds an integer value
int *sumPtr;                // sumPtr holds an address where an integer can be found
```

By now there may be plenty of confusion between the symbols * and & so we now discuss their use.

Using the & Symbol

The & symbol is basically used on two occasions:

1. The most frequent use we have seen is between the data type and the variable name of a pass by reference parameter in a function heading/prototype. This is called a **reference variable**. The memory address of the argument (parameter) is sent to the function instead of the value at that address. When the parameter is used in the function, the compiler automatically **dereferences** the variable. Dereference means that the location of that reference variable (parameter in this case) is accessed to retrieve or store a value.

   We look at a swap function (a function that swaps the value of two numbers) to show that the `&` symbol is used in the parameters that need to be swapped. The reason is that these values need to be changed by the function and thus, we give the address (location in memory) of those values so that the function can write their new values into them as they are swapped.

Example:10.1

```
void swap (int &first, int &second)          //The & indicates that the parameters first and second
                                             //are being passed by reference
{
        int temp;
        temp = first;                 //since first is a reference variable, the
                                      //compiler retrieves the value stored there and places it in temp
        first = second;               // New values are written directly into the
        second = temp;                // the memory locations of first and second
}
```

2. The & symbol is also used whenever we are interested in the **address** of a variable rather than its **contents**.

```
cout << sum;                  // This outputs the value stored in the variable sum
cout <<&sum;                  // This outputs the address where sum is stored in memory
```

   Using the `&` symbol to get the address of a variable comes in handy when we are assigning values to pointer variables.

Using the * Symbol

The `*` symbol is also basically used on two occasions.

1. It is used to define pointer variables.
   ```
   int *ptr;
   ```

2. It is also used whenever we are interested in the contents of the memory location pointed to by a pointer variable, rather than the address itself. When used this way `*` is called the i**ndirection operator** or **dereferencing operator**.

Example:

```
cout << *ptr;     // Since ptr is a pointer variable, * dereferences ptr.  The value stored at
                  // the location ptr points to will be printed.
```

Using * and & Together

In many ways `*` and `&` are the opposite of each other. The `*` symbol is used just before a pointer variable so that we may obtain the actual data rather than the address of a variable. The `&` symbol is used on a variable so that the variable's address, rather than the data stored in it, will be used. The following program demonstrates the use of pointers.

Example 10.2

```
#include <iostream>
using name space std;
int main()
{
        int one = 10;
        int* ptr1;        // ptr1 is a pointer variable that points to an int

        ptr1 = &one; //&one indicates that the address, not contents, of one is being assigned to ptr1.
                     // Remember that ptr1, can only hold an address.  Since ptr1 holds the address
                     // where the variable one is stored, we say that ptr1 "points to" one.

        cout  <<   "The value of one is " << one << endl << endl;
        cout  <<  "The value of &one is " <<  &one  <<  endl << endl;
        cout  <<  "The value of ptr1 is " <<  ptr1  <<  endl << endl;
        cout  <<  "The value of *ptr1 is " << *ptr1  << endl << endl;
        return 0;
}
```

What do you expect will be printed if the address of variable one is the hexadecimal value 006AFOF4?

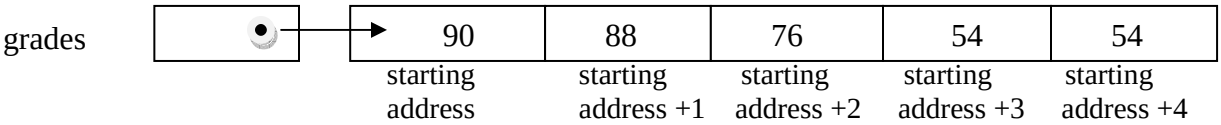
The following will be printed by the program:

Output	Comments
The value of one is 10	one is an integer variable, holdina a 10.
The value of &one is 006AF0F4	**`&one`** is the "address of" variable one
The value of ptr1 is 006AF0F4	**`ptr1`** is assigned one's address
The value of *ptr1 is 10	`*` is the dereferencing operator which means **`*ptr1`** gives us the value of the variable `ptr1` is pointing at.

Arrays and Pointers

When arrays are passed to functions they are passed by pointer. An array name is a pointer to the

beginning of the array. Variables can hold just one value and so we can reference that value by just naming the variable. Arrays, however, hold many values. All of these values cannot be referenced just by naming the array. This is where pointers enter the picture. Pointers allow us to access all the array elements. Recall that the array name is really a pointer that holds the address of the first element in the array. By using an array index, we dereference the pointer which gives us the contents of that array location. If `grades` is an array of 5 integers, as shown below, `grades` is actually a pointer to the first location in the array, and `grades[0]` allows us to access the contents of that first location.



From the last section we know it is also possible to dereference the pointer by using the * operator. What is the output of the following two statements?

```
cout << grades [0];        // Output the value stored in the 1st array element
cout << *grades;           // Output the value found at the address stored in grades
                           // (i.e., at the address of the 1st array element)
```

Both statements are actually equivalent. They both print out the contents of the first `grades` array location: a 90.

Access of an individual element of an array through an index is done by pointer arithmetic. We can access the second array location with `grades[1],` the third location with `grades[2],` and so on, because the indices allows us to move through memory to other addresses relative to the beginning address of the array. The phrase "address + 1" in the previous diagram means to move one array element forward from the starting address of the array. The third element is accessed by moving 2 elements forward and so forth. The amount of movement in bytes depends on how much memory is allocated for each element, and that depends on how the array is defined. Since `grades` is defined as an array of integers, if an integer is allocated 4 bytes then + 1 means to move forward 4 bytes from the starting address of the array, +2 means to move foreward 8 bytes etc. The compiler keeps track of how far forward to move to find a desired element based on the array index. Thus the following two statements are equivalent.

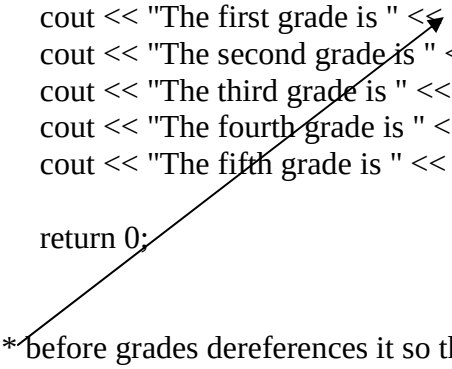
```
cout << grades[2];
cout << *(grades +2);
```

Both statements refer to the value located two elements forward from the starting address of the array. Although the first may be the easiest, computer scientists need to understand how to access memory through pointers. The following program illustrates how to use pointer arithmetic rather than indexing to access the elements of an array.

Example 10.3

```
// This program illustrates how to use pointer arithmetic to access elements of an array
```

```
#include <iostream >
using namespace std;

int main()
{
        int grades[5] = {90, 88, 76 ,54, 34};
                // This defiens and initializes an int array.
                // Since grades is an array name, it is really a pointer that holds
                // the starting address of the arry
        cout << "The first grade is " << *grades << endl;
        cout << "The second grade is " << *(grades + 1) << endl;
        cout << "The third grade is " << *(grades + 2) << endl;
        cout << "The fourth grade is " << *(grades + 3)<< endl;
        cout << "The fifth grade is " << *(grades + 4) << endl;

        return 0;
}
```



The * before grades dereferences it so that the contents of array location 0 is printed instead of its address. The same is done for the other elements of the array. In each case, pointer arithmetic gives us the address of the next array element. Then the indirection operator * gives us the value of what is stored at that address.

What is printed by the program?

The first grade is 90
The second grade is 88
The third grade is 76
The fourth grade is 54
The fifth grade is 34

Dynamic Variables

The size of an array has to be given at the time of its definition. The programmer must estimate the maximum number of elements that will be used by the array and this size is static, i.e., it cannot change during the execution of the program. Consequently, if the array is defined to be larger than is needed, memory is wasted. If it is definedto be smaller then is needed, there is not enough memory to hold all the elements. The use of pointers (and the **`new`** and **`delete`** operators described below) allows us to dynamically allocate enough memory for an array so that memory is not wasted.

This leads us to **dynamic variables**. Pointers allow us to use dynamic variables, which can be created and destroyed as needed within a program. We have studied scope rules, which define where a variable is active. Related to this is the concept of **lifetime,** the time during which a variable exists. The lifetime of dynamic variables is controlled by the program through explicit commands to allocate (i.e., create) and deallocate (i.e., destroy) them. The operator **`new`** is used to allocate and the operator

delete is used to deallocate dynamic variables. The compiler keeps track of where in memory non-dynamic variables (variables discussed thus far) are located. Their contents can be accessed by just naming them. However, the compiler does not keep track of the address of a dynamic variable. When the new commad is used to allocate memory for a dynamic variable, the system returns its address and the programmer stores it in a pointer variable. Throught the pointer variable we can access the memory location.

Example:

```
int *one;              //one and two are defined to be pointer variables that point to ints.
int *two;

int result;             //defines an int variable that will hold the sum of two values

one = new int;         //These statements each dynamically allocate enough memory to hold
 two = new int;        // an int and assign their addresses to pointer variables one and two
                       // respectively.

*one = 10;             // These statements assign the value of 10 to the memory location
                       //pointed to by one and 20 to the memory location pointed to by two.
*two = 20;

result = *one + *two;   // This adds the contents of the memory locations pointed to by one and two


cout << "result = " << result << endl;

delete one;            //These statements deallocate the dynamic variables.  Their memory is
delete two;            //freed and they cease to exist.
```

Now let us use dynamic variables to allocate an appropriate amount of memory to hold an array. By using the new operator to create the array, we can wait until we know how big the array needs to be before creating it. The following program demonstrates this idea. First the user is asked to input the number of grades to be processed. That number is then used to allocate exactly enough memory to hold an array with the required number of elements for the grades.

Example 10.4

```
// This program finds the average of grades
// It dynamically allocates space for the array holding the grades

#include <iostream>
using namespace std;
```

```
// function prototypes

void sortIt(float* grades, int numOfGrades);    //prototype of function to sort
void displayGrades (float* grades, int numOfGrades);

int main()
{
        float *grades;                  // a pointer that will be used to point to the beginning of
                                        //  a float array
        float total = 0;                //  total of all grades
        float average;                  //  average of all grades
        int numOfGrades;                // The number of grades to be processed
        int count;                      //loop counter

        cout << fixed << showpoint << setprecision(2);

        cout << "How many grades will be processed " << endl;
        cin  >>  numOfGrades;

        while (numOfGrades <= 0)                // checks for a legal value
        {
                cout << “There must be at least one grade.  Please reenter.\n”;
                cout << “How many grades will be processed “ << endl;
                cin >>  numOfGrades
        }

        grades = new float[numOfGrades];            //allocate memory for an array
                                                    // new is the operator that is allocating an array of
                                                    // floats with the number of elements specified by
                                                    // the user.  grades is the pointer holding the
                                                    // starting address of the array

        if (grades == NULL)                          //Null is a special identifier predefined to equal 0.
                                                    // It indicates a non-valid address.  If grades is 0
                                                    // it means the operating system was unable to
                                                    // allocate enough memory for the array.
        {
                cout  <<  "Error allocating memory!\n";
                        // The program should output an appropriate error message and return with a
                         // value other than 0 to signal a problem
                        return - 1;
         }

        cout<< "Enter the grades below\n";

        for ( count = 0; count < numOfGrades; count++)
```

```
	{
		cout << "Grade " << (count+1) << ":" << endl;
		cin >> grades[count];
		total = total + grades[count];
	}

	average = total / numOfGrades;
	cout  << "Average Grade is " <<  average <<  "%" << endl;

	sortIt(grades, numOfGrades); //call to a sort function
	delete [] grades; //deallocates array memory the []
	return 0;
}

//*************************************************************************
//                              sortIt
//
//  task:               to sort numbers in an array
//  data in:            an array of floats and the number of elements in the array
//
//  data out:           sorted array
//
//*************************************************************************


void Sortit(float* grades, int numOfGrades)
{

	// Sort routine placed here
}

//*************************************************************************
//                              displayGrades
//
//  task:               to display numbers in an array
//  data in:            an array of floats and the number of elements in the array
//
//  data out:           none
//
//*************************************************************************

void displayGrades (float* grades, int numOfGrades)
{
	// Code to display grades of the array
}
```

Notice how the dynamic array is passed as a parameter to the `sortIt` and `displayGrades` functions. In each case, the call to the function simply passes the name of the array, along with its size as an argument. The name of the array holds the array's starting address.

```
sortIt(grades, numOfGrades);
```

In the function header, the formal parameter that receives the array is defined to be a poiner data type.

void sortIt(float* grades, int numOfGrades)

Since the compiler treats an array name as a pointer, we could also have written the following function header as:

void sortIt(float grade[], int numOfGrades)

In the program, dynamic memory was used to save memory. This is a minor consideration for the type of programs done in this course, but a major concern in professional programming environments where large fluctuating amounts of data are used.

Review of * and &

The * symbol is used to define pointer variables. In this case it appears in the variable definition statement between the data type and the pointer variable name. It indicates that the variable holds an address, rather than the data stored at that address.

Example 1: `int * ptr1;`

* is also used as a dereferencing operator. When placed in front of an already defined pointer variable, the data stored at the location the pointer points to will be used and not the address.

Example 2: `cout << *ptr1;`

Since `ptr1` is defined as a pointer variable in Example 1, if we assume `ptr1` has now been assigned an address, the output of Example 2 will be the data stored at that address. `*` in this case dereferences the variable `ptr1`.

The & symbol is used in a procedure (method) or function heading to indicate that a parameter is being passed by reference. It is placed between the data type and the parameter name of each parameter that is passed by reference.

The & symbol is also used before a variable to indicate that the address, not the contents, of the variable is to be used.

Example 3:

int *ptr1;
int one = 10;

```
ptr1 = &one;                    //This assigns the address of variable one to ptr1

cout  << "The value of &one is " << &one << endl;          //This prints an address

cout << "The value of *ptr1 is " << *ptr1 << endl;   //This prints 10, because ptr1 points to one and *
                                                     // is the dereferencing operator
```

Pointers and Strings

Pointers can be very useful for writing string processing functions. If one needs to process a certain string, the beginning address can be passed with a pointer variable. The length of the string does not even need to be known since the computer will start processing using the address and continue through the string until the null character is encountered.

Sample Program 10.5 below reads in a string of no more than 50 characters and then counts the number of letters, digits, and whitespace characters in the string. Notice the use of the pointer `strPtr`, which points to the string being processed. The three functions `countLetters`, `countDigits`, and `countWhiteSpace` all perform basically the same task—the `while` loop is executed until `strPtr` points to the null character marking the end of the string. In the`countLetters` function, characters are tested to see if they are letters. The `if(isalpha(*strPtr))`statement determines if the character pointed at by `strPtr` is a letter. If so, then the counter `occurs` is incremented by one. After the character has been tested, `strPtr` is incremented by one to test the next character. The other two functions are analogous.

Example Sample Program 10.5:

```
#include <iostream>
#include <ctype.h>
using namespace std;

//function prototypes

int countLetters(char*);
int countDigits(char*);
int countWhiteSpace(char*);

int main()
{
        int numLetters, numDigits, numWhiteSpace;
        char inputString[51];

        cout << "Enter a string of no more than 50 characters: " << endl << endl;
        cin.getline(inputString,51);

        numLetters = countLetters(inputString);
```

```
	numDigits = countDigits(inputString);
	numWhiteSpace = countWhiteSpace(inputString);

	cout << "The number of letters in the entered string is " <<  numLetters << endl;
	cout << "The number of digits in the entered string is " <<  numDigits << endl;
	cout << "The number of white spaces in the entered string is " << numWhiteSpace << endl;

	return 0;
}

//*************************************************************************
//                                  countLetters
//
//  task:               This function counts the number of letters (both upper and lower
//                      case) in the string
//  data in:            pointer that points to an array of characters
//  data returned:      number of letters in the array of characters
//
//*************************************************************************

int countLetters(char *strPtr)
{
	int occurs = 0;
	while(*StrPtr != '\0')          //  loop is exectued as long as the pointer strPtr
                                        // does not point to the null character which
                                        //  marks the end of the string
	{
		if (isalpha(*strPtr))   // isalpha determines if the character is a letter
		occurs++;
		strPtr++;
	}
	return occurs;
}


//*************************************************************************
//                                  countDigits
//
//  task:               This function counts the number of digits in the string
//  data in:            pointer that points to an array of characters
//  data returned:      number of digits in the array of characters
//
//*************************************************************************

int countDigits(char *strPtr)
{
```

```
        int occurs = 0;
        while(*strPtr != '\0')
        {
                if (isdigit(*StrPtr))           //  isdigit determines if the character is a digit
                        occurs++;
                strPtr++;
        }
        return occurs;
}


//*************************************************************************
//                                      countWhiteSpace
//
//  task:               This function counts the number of whitespace characters
//                      in the string  This includes space, newline, vertical tab and tab
//  data in:            pointer that points to an array of characters
//  data returned:      number of whitespaces in the array of characters
//
//*************************************************************************

int countWhiteSpace(char *StrPtr)
{
        int occurs = 0;
        while(*strPtr != '\0')
        {
                if (isspace(*strPtr))           //  isspace determines if the character is a
                                                //  whitespace character
                   occurs++;
                strPtr++;
}
return occurs;
}
```

Lab 10: Pointers

The purpose of this lab is to introduce pointer variables and their relationship with arrays, the dereferencing operator and the concept of dynamic memory allocation.

The lab begins with a quiz on the material covered in this lesson.

Fill-in-the-Blank Questions

1. The_____symbol is the dereferencing operator.

2. The _____symbol means "address of."

3. The name of an array, without any brackets, acts as a(n)_____
   to the starting address of the array.

4. An operator that allocates a dynamic variable is _____.

5. An operator that deallocates a dynamic variable is _____.

6. Parameters that are passed by_____ is similar to pointer variables in that they can contain the address of another variable. They are used as parameters of a procedure (void function) whenever we want a procedure to change the value of the argument.

Given the following information fill the blanks with either "an address" or "3.75".

```
float * pointer;
float pay = 3.75;
pointer = &pay;
```

7. cout << pointer; will print_____ .

8. cout << *pointer; will print_____ .

9. cout << &pay; will print _____.

10. cout << pay; will print_____ .

Problem 10.1 Introduction to Pointer Variables

Retrieve program `pointers.cpp` from the Lab 10 folder. The code is as follows:

```
// This program demonstrates the use of pointer variables
// It finds the area of a rectangle given length and width
// It prints the length and width in ascending order
```

```
// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

int main()
{

   int length;             // holds length
   int width;              // holds width
   int area;               // holds area

   int *lengthPtr;         // int pointer which will be set to point to length
   int *widthPtr;           // int pointer which will be set to point to width

   cout << "Please input the length of the rectangle" << endl;
   cin >> length;
   cout << "Please input the width of the rectangle" << endl;
   cin >> width;

   // Fill in code to make lengthPtr point to length (hold its address)
   // Fill in code to make widthPtr point to width (hold its address)


   area =  // Fill in code to find the area by using only the pointer variables
   cout << "The area is " << area << endl;



   if (// Fill in the condition of length > width by using only the pointer variables)
         cout << "The length is greater than the width" << endl;

   else if (// Fill in the condition of width > length by using only the pointer
               // variables)
         cout << "The width is greater than the length" << endl;

   else
         cout << "The width and length are the same" << endl;


   return 0;
}
```

Exercise 1: Complete this program by filling in the code (places in bold). Note: use only pointer variables when instructed to by the comments in bold. The program is to test your knowledge of pointer variables and the & and * symbols.

Exercise 2: Run the program with the following data: 10 15.
Record the output here_____

Problem 10.2 Dynamic Memory

Retrieve program `dynamic.cpp` from the Lab 10 folder.
The code is as follows:

```
// This program demonstrates the use of dynamic variables
// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

const int MAXNAME = 10;

int main()

{
  int pos;
  char * name;
  int * one;
  int * two;
  int * three;
  int result;

  //  Fill in code to allocate the integer variable one here
  //  Fill in code to allocate the integer variable two here
  //  Fill in code to allocate the integer variable three here
  //  Fill in code to allocate the character array pointed by name

  cout << "Enter your last name with exactly 10 characters." << endl;
  cout << "If your name has < 10 characters, repeat last letter. " << endl
                << "Blanks at the end do not count." << endl;

  for ( pos = 0;  pos < MAXNAME;   pos++)

        cin >> // Fill in code to read a character into the name array
               // WITHOUT USING a bracketed subscript

  cout << "Hi ";
  for ( pos = 0;  pos < MAXNAME;  pos++)
```

```
    cout <<       // Fill in code to a print a character from the name array
                  // WITHOUT USING a bracketed subscript

  cout << endl << "Enter three integer numbers separated by blanks" << endl;

  // Fill in code to input three numbers and store them in the
  // dynamic variables pointed to by pointers one, two, and three.
  // You are working only with pointer variables

  //echo print
  cout << "The three numbers are " << endl;

  // Fill in code to output those numbers

  result = // Fill in code to calculate the sum of the three numbers
  cout << "The sum of the three values is " << result << endl;

 // Fill in code to deallocate one, two, three and name



  return 0;
}
```

Exercise 1: Complete the program by filling in the code. (Areas in bold)
This problem requires that you study very carefully the code already written to prepare you to complete the program.

Sample Run:

Enter your last name with exactly 10 characters.
If your name < 10 characters, repeat last letter. Blanks do not count.
DeFinooooo
Hi DeFinooooo
Enter three integer numbers separated by blanks
5 6 7
The three numbers are 5 6 7
The sum of the three values is 18

Exercise 2: In inputting and outputting the name, you were asked NOT to use a bracketed subscript. Why is a bracketed subscript not necessary?

Would using `name[pos]` work for inputting the name? Why or why not?
Would using `name[pos]` work for outputting the name? Why or why not?
Try them both and see.

Problem 10.3 Dynamic Arrays

Retrieve program `darray.cpp` from the Lab 10 folder.
The code is as follows:

```
// This program demonstrates the use of dynamic arrays

// PLACE YOUR NAME HERE

#include <iostream>
#include <iomanip>
using namespace std;


int main()

{

  float  *monthSales;          // a pointer used to point to an array
                               // holding monthly sales
  float  total = 0;            // total of all sales
  float  average;              // average of monthly sales
  int    numOfSales;           // number of sales to be processed
  int    count;                // loop counter

  cout << fixed << showpoint << setprecision(2);

  cout << "How many monthly sales will be processed? ";
  cin >> numOfSales;

  // Fill in the code to allocate memory for the array pointed to by
  // monthSales.


  if ( // Fill in the condition to determine if memory has been
       // allocated (or eliminate this if construct if your instructor
       // tells you it is not needed for your compiler)
     )

  {
          cout << "Error allocating memory!\n";
          return 1;
  }

  cout << "Enter the sales below\n";
```

```
  for ( count = 0; count < numOfSales;  count++)
        {
     cout << "Sales for Month number  "
                  << // Fill in code to show the number of the month
                        << ":";

          // Fill in code to bring sales into an element of the array
        }

  for ( count = 0; count < numOfSales; count++)
        {
          total = total + monthSales[count];
        }

  average = // Fill in code to find the average


  cout << "Average Monthly sale  is $" << average << endl;
  // Fill in the code to deallocate memory assigned to the array.

  return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold.

Sample Run:

How many monthly sales will be processed 3
Enter the sales below
Sales for Month number 1: 401.25
Sales for Month number 2: 352.89
Sales for Month number 3: 375.05
Average Monthly sale is $376.40

Student Generated Code Assignments

In these assignments you are asked to develop functions that have dynamic arrays as parameters. Remember that dynamic arrays are accessed by pointer variables and thus the parameters that serve as dynamic arrays are, in fact, pointer variables.

Example:

```
void sort(float* score, int num_scores);      // a prototype whose function has adynamic array as its
                                              // first parameter. It is a pointer variable
```

```
..
int main();
{
        float *score;                   // a pointer variable
        ..
        ..
        score = new float[num_scores];  // allocation of the array

        sort(score,scoreSize);          // call to the function
```

Student Generated Code Problem 10.4

Write a program that will read scores into an array. The size of the array should be input by the user (dynamic array). The program will find and print out the average of the scores. It will also call a function that will sort (using a bubble sort) the scores in ascending order. The values are then printed in this sorted order.

Sample Run:
Please input the number of scores
5
Please enter a score
100
Please enter a score
90
Please enter a score
95
Please enter a score
100
Please enter a score
90
The average of the scores is 95

Here are the scores in ascending order
90
90
95
100
100

Student Generated Code Problem 10.5

This program will read in id numbers and place them in an array. The array is dynamically allocated large enough to hold the number of id numbers given by the user. The program will then input an id and call a function to search for that id in the array. It will print whether the id is in the array or not.

Sample Run:

Please input the number of id numbers to be read
4
Please enter an id number
96
Please enter an id number
97
Please enter an id number
98
Please enter an id number
99
Please input an id number to be searched
67
67 is not in the array

Student Generated Code Problem 10.6

Write a program that will read monthly sales into a dynamically allocated array. The program will input the size of the array from the user. It will call a function that will find the yearly sum (the sum of all the sales). It will also call another function that will find the average.

Sample Run:

Please input the number of monthly sales to be input
4
Please input the sales for month 1
1290.89
Please input the sales for month 2
905.95
Please input the sales for month 3
1567.98
Please input the sales for month 4
994.83
The total sales for the year is $4759.65
The average monthly sale is $1189.91

Lesson 10: Summary

A **pointer variable** holds an address of the location of the variable rather than data. called **pointers,** which hold these addresses.

To define a variable to be a pointer, we precede it with an asterisk (*):
```
int *ptr;
```

The address of a non-pointer variable is given by preceding the variable name with the C++ address operator (**&**):

```
cout  <<  &sum;   // This outputs the address of the variable sum and not the contents of the memory  location.
```

The (&) operator in front of a variable `sum` indicates that the address itself, not the data stored in that location, is the value used.

```
int sum;                    // sum holds an integer value
int *sumPtr;                // sumPtr holds an address where an integer can be found
```

The & symbol is basically used on two occasions:

1. The most frequent use we have seen is between the data type and the variable name of a pass by reference parameter in a function heading/prototype. This is called a **reference variable**. The memory address of the argument (parameter) is sent to the function instead of the value at that address. When the parameter is used in the function, the compiler automatically **dereferences** the variable. Dereference means that the location of that reference variable (parameter in this case) is accessed to retrieve or store a value.

2. The & symbol is also used whenever we are interested in the **address** of a variable rather than its **contents**.

```
cout << sum;                // This outputs the value stored in the variable sum
cout <<&sum;                // This outputs the address where sum is stored in memory
```

The * symbol is also basically used on two occasions.

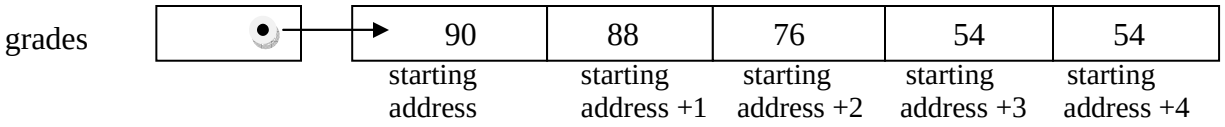
1. It is used to define pointer variables.
   ```
   int *ptr;
   ```

2. It is also used whenever we are interested in the contents of the memory location pointed to by a pointer variable, rather than the address itself. When used this way * is called the i**ndirection operator** or **dereferencing operator**.

Pointers allow us to use **dynamic variables**, which can be created and destroyed as needed within a program. The **lifetime** of dynamic variables is controlled by the program through the explicit commands `new` to allocate (i.e., create) and `delete` deallocate (i.e., destroy) them.
Example:

```
int *one;                 //one and two are defined to be pointer variables that point to ints.
int *two;
int result;               //defines an int variable that will hold the sum of two values
one = new int;            //These statements each dynamically allocate enough memory to hold
 two = new int;           // an int and assign their addresses to pointer variables one and two  respectively.
*one = 10;                // These statements assign the value of 10 to the memory location
                          //pointed to by one and 20 to the memory location pointed to by two.
*two = 20;
result = *one + *two;   // This adds the contents of the memory locations pointed to by one and two
cout << "result = " << result << endl;
delete one;               //These statements deallocate the dynamic variables.  Their memory is
delete two;               //freed and they cease to exist.
```

Arrays are stored as a pointer variable




```
cout << grades [0];       // Output the value stored in the 1st array element
cout << *grades;          // Output the value found at the address stored in grades
                          // (i.e., at the address of the 1st array element)
```

Lesson 11: More about Classes and Operation Overloading

Static Member Variables

Each object has its own copy of the class member variables. This means that two objects of the same class will likely have different values for a given member variable. For example, two savings accounts may have different account numbers and balances. Look at Example program 11.1

```
#include <iostream>
using namespace std;

class SavingsAcct
{
        private:
                int accNum;                     //member variables
                double balance;

        public:
                SavingsAcct();                  // default constructor
                SavingsAcct(int, double);
                void newAcctInfo();
                void displayAcctInfo();
                void deposit(double);
                void withdraw(double);
};
// Member Function Implementation Section

SavingsAcct::SavingsAcct()                      // default constructor
{
        acctNum=0;
        balance =0;
        newAcctInfo();
}

SavingsAcct::SavingsAcct(int num, double startBal)    // constructor
{
        accNum=num;
        balance = startBal;
        newAcctInfo();
}

void SavingsAcct::newAcctInfo()
{
        cout <<  "new account number: " << acctNum "   Initial Balance: $"
              <<  balance << endl;
}
// Client code
```

```
int main()
{
        SavingsAcct acct1(1001,500);
        SavingsAcct acct2(1002,350);

        // other code

        return 0;
}
```

Sample run

```
New account number:  1001    Initial Balance: $500
New account number:  1002    Initial Balance: $350
```

There are times however when we would like all class objects to share, or have access to a common copy of a member variable. For example a member variable that keeps track of the total balance of all accounts. We can do accomplish this by using **static member variables**.

A static member variable is declared by placing the word static in front of it in the class declaration and then placing a separate definition of the variable outside the class. The definition itself does not include the word static, but does include the class name and scope resolution operator (::), just like class member function implementations do. As with any variable, an initial value can be given to the static member variable at the time of its definition. The variable exists for the entire duration of the program that uses the class and is accessible to all instance of the class.

Example 11.2 modifies Example 11.2 to include two static member variables. Changes to the earlier sample program are noted in bold.

```
#include <iostream>
using namespace std;

class SavingsAcct
{
        private:
                int accNum;                     //member variables
                double balance;
                static int nextNumber;          // static member variables
                static double totalBalances;




        public:
                SavingsAcct();                  // default constructor
```

```
            SavingsAcct(double);            // constructor
            void newAcctInfo();
            double getTotalBalances();
            void displayAcctInfo();
            void deposit(double);
            void withdraw(double);
};

// Static Variable Definitions—Place these just before the member function Implementation
// Section

int     SavingsAcct::nextNumber = 1001;     // first number to be given out
double SavingsAcct::totalBalances =0;       // start static accumulator at 0

// Member Function Implementation Section

SavingsAcct::SavingsAcct()                  // default constructor
{
        acctNum=0;
        balance =0;
        newAcctInfo();
}

SavingsAcct::SavingsAcct(int num, double startBal)    // constructor
{
        accNum=nextNumber;                  // gives out the next available acct number
        nextNumber++;                       // increments next available acct number
        balance = startBal;
        totalBalances –=totalBalances +balance;   // increments totalBalances by new
                                            // starting  balance
        newAcctInfo();
}

void SavingsAcct::newAcctInfo()
{
        cout <<  “new account number: “ << acctNum “   Initial Balance: $”
             <<  balance << endl;
}


double SavingsAcct::getTotalBalances()
{
        return totalBalances;
}
// Client code
```

```
int main()
{
        SavingsAcct acct1(500);   // Objects are now created with just one parameter:
        SavingsAcct acct2(350);   // starting balance

        // The next statement invokes acct1's getTotalBalances() member function to
        // return totalBalances.  We could just as easily have invoked it through acct2
        // since they both have access to this common member variable.

        cout << "Total savings deposits = $" << acct1.getTotalBalances() << endl;
        // other code

        return 0;
}
```

Static Member Functions

A **static member function** is a special function that is normally used to initialize static member variables or to perform other setup tasks for a class. Since Static member variables exist for the entire duration of a program, they exist even before any instances of the class are created. Static member functions also exist separately from individual class objects and can be called even before any instances of the class are created. As a result, they can only access static member variables, which can be initialized at the time they are defined or be static member functions. Static member variables should not be initialized by constructors, since they are initialized only once and not each time an instance of the class is created. A static member function is declared by placing the word `static` in front of its prototype within the class declaration. In the following example `initGoal` is a static function belonging to a class named `SalesForce`. The prototype is:

```
static void initGoal(double);  //  used to initialize a static member variable named
                               //  salesGoal of the SalesForce class
```

The function implementation would be identical to that of any regular member function.

```
void SalesForce::initGoal(double amt)
{
      salesGoal = amt;
}
```

There is a difference in the way static member functions are called. Since static member functions exist separate from class objects and can be called before any class objects even exist, they are not called by naming a class object. Instead they are called by preceding the function name with the name of the class and the scope resolution operator.

Ex.

```
SalesForce::initGoal(15000);  // This is a call to the static function initGoal to initialize
                              // salesGoal to 15000
```

Example Program 11.3

```
//  This program introduces the use of static member functions.
//  It also illustrates passing an object by reference to a function

#include <iostream>
using namespace std;

class SalesForce
{
        private:
                static double salesGoal;        //  static variable shared by all class objects
                int snum;                       // sales rep. number
                double monthlySales;
        public:
                static void initGoal(double);   // static function to set salesGoal

                SalesForce() {  snum = monthlySales = 0;)  // default constructor
                SalesForce (int, double);       // constructor
                void displayInfo();             //  displays an object's member data
                bool metGoal();                 //  indicates if sales goal has been met
};

// Static Member Variable Definitions

double SalesForce::salesGoal;

//  Member Function Implementation Section

void SalesForce::initGoal(double amt)
{
        salesGoal = amt;
}

SalesForce::SalesForce(int num, double amt)  // constructor
{
        snum = num;
        monthlySales = amt;
}

void SalesForce::displayInfo()
{
        cout << "Sales Rep Number: "  < snum << " Total sales $" << monthlySales
```

```
                << endl;
}

bool SalesForce::metGoal()
{
        if (montlySales >= salesGoal)
                return true;
        else
                return false;
}

// Client Code
//  Function prototypes
void salesReport (SalesForce &);

int main()
{
        double goalAmount;

        cout << "Input monthly sales goal: ";
        cin  >> goalAmount;
        SalesForce::initGoal(goalAmount);   // set the sales goal for the class
        SalesForce repOne (101, 13500);      // create 1st sales rep.
        SaleForce repTwo(202,16,800);        // create 2nd rep.

        salesReport(repOne);
        salesReport(repTwo);

        return 0;
}

void saleReport(SalesForce & rep)
{
        rep.displayInfo();
        if (rep.metGoal())
                cout << "Met sales goal\n\n";
        else
                cout << "Did not meet sales goal\n\n";
}
```

Friend Functions

A **friend function** is a function that is not a member of a class, but which has access to the private members of the class. Normally private member variables and member functions are hidden from all parts of a program outside the class. This safeguards member variables as well as shields outside functions from implementation details they do not need to know. For a function to access a private

member variable, it normally must do so through a call to a public member function which can carefully control how the variable is accessed. Friend functions allow us to make an occasional exception to this protocol. They can be standalone functions or members of another class. In fact, even an entire class can be declared to be a friend of another class.

Classes declare their friends. That is, in order to have "friend" status, a function must be granted that status by the class whose private member it will be allowed to access. A friend function is declared by placing its prototype preceded by the key word `friend`, inside the class declaration of the class granting the friend status. Strictly speaking, friend functions are neither private nor public, so they could be placed in either section of the class declaration. However, it is most common to place them in the public section. For example, the `SalesForce` class of the last section could declare a stand-alone function named salesReport to be a friend function by including the following prototype within its class declaration:

```
friend void salesReport(SalesForce &);
```

If instead the salesReport function were a member of another class, let us call it SalesOffice, it would be declared a friend function with the following declaration:

```
friend void SalesOffice::salesReport(SalesForce &);
```

Either way, this indicates that the `salesReport` is not a member of the `SalesForce` class, but rather an outside function, declared elsewhere, which can access `SalesForce` private members.

Notice that the `salesReport` function must receive as a parameter a reference to an object of the class whose private members it is being allowed to access. In the program where the `salesReport` function is defined, this parameter must appear in the prototype and the function heading. However, the keyword friend will not appear in either since a function cannot declare itself to be a friend. The `salesReport` function heading would look like the following:

```
void salesReport(SalesForce & salesRep)
```

It would be called as follows, with some `SalesForce` object passed to it:

```
salesReport(repOne);
```

It could then access either static or regular data members of the object passed to it by using the dot operator, as if it were accessing a data member of a simple structure.

```
cout << salesRep.monthlySales << endl;
cout << salesRep.salesGoal << endl;
```

Example program 11.4 modifies example program 11.3 to use a friend function. Changes are in bold type.

Example program 11.4

// This program introduces the use of friend functions

```
#include <iostream>
using namespace std;

class SalesForce
{
        private:
                static double salesGoal;        //  static variable shared by all class objects
                int snum;                       // sales rep. number
                double monthlySales;
        public:
                static void initGoal(double);   // static function to set salesGoal

                SalesForce() {  snum = monthlySales = 0;)  // default constructor
                SalesForce (int, double);       // constructor
                void displayInfo();             //  displays an object's member data
                bool metGoal();                 //  indicates if sales goal has been met

                friend void salesReport(SalesForce &);  // declares a friend function
};

// Static Member Variable Definitions

double SalesForce::salesGoal;

//  Member Function Implementation Section

void SalesForce::initGoal(double amt)
{
        salesGoal = amt;
}

SalesForce::SalesForce(int num, double amt)  // constructor
{
        snum = num;
        monthlySales = amt;
}

void SalesForce::displayInfo()
{
        cout << "Sales Rep Number: "  < snum << " Total sales $" << monthlySales
             << endl;
}

bool SalesForce::metGoal()
{
```

```
        if (montlySales >= salesGoal)
                return true;
        else
                return false;
}

// Client Code
//  Function prototypes
void salesReport (SalesForce &);

int main()
{
        double goalAmount;

        cout << "Input monthly sales goal: ";
        cin  >> goalAmount;
        SalesForce::initGoal(goalAmount);   // set the sales goal for the class
        SalesForce repOne (101, 13500);     // create 1st sales rep.
        SaleForce repTwo(202,16,800);       // create 2nd rep.

        salesReport(repOne);
        salesReport(repTwo);

        return 0;
}

//*************************************************************************
//                              salesReport
//
//  Revised salesReport function to produce a sales report.  Now that the SalesForce
//  class has declared this a friend function, it can access private members of SalesForce
//  class objects directly by using the dot operator, instead of  through calls to public
//  member functions.
//*************************************************************************


void saleReport(SalesForce & rep)
{
        cout << "Report for Sales Rep. number: " << rep.snum << endl;
        cout << "Monthly sales: $" << rep.monthlySales
             << "      Goal: $" << rep.salesGoal << endl;
        if (rep.metGoal())
                cout << "Met sales goal\n\n";
        else
                cout << "Did not meet sales goal\n\n";
}
```

The risk in using friend functions is that once given friend status they can access any private data of the class object they are passed, including data they do not need in order to complete their tasks. Nothing is really safeguarded from a friend function. For this reason, most programmer greatly limit their use.

Memberwise Assignment

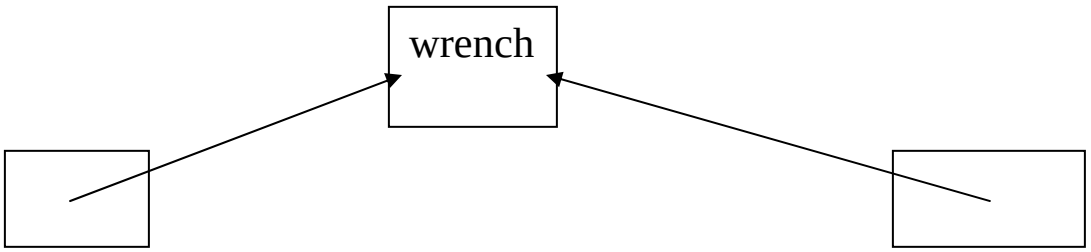
```
item1 = item2;   // This copies the data values of one existing
                 // Inventory object to another
```

This data copy from one object to another is called **memberwise assignment.** Memberwise assignment can also be used to initialize an object when it is created. The following statement would create a new Inventory object, `item2`, and initialize it member variables to the value of `item2`.s corresponding member variables.

```
Inventory item3 = item2;  //   This creates a new inventory object and
                          //   initializes its data values with
                         //    another Inventory object's data.
```

Problems with Memberwise Assignment

Memberwise assignment provides a convenient way to set one object's data equal to another's or to initialize an object when it is created. In fact, if we pas an object to a function using pass by value, this is what automatically occurs. A new local object is created and it is initialized, using a memberswise copy, with the data of the object that was passed to the function. However, in certain circumstances, as when a pointer to dynamic memory is involved, memberwise copy causes a problem. This is because we want the new object to have its own memory space for its variable and just be initialized with the data being copied. Instead, memberwise copy would give the new object the address of the old object's data member. The following diagram illustrates this. Assume item1 and item2 are two objects of a class with a data member called description. Description is a pointer to dynamic memory where the description information is stored. The statement `item1 = item2;` would have the following result:



item1.description item2.description

This is not what we want to happen. Any change item1 or item2 now makes to its description variable would change the description of the other object as well. Some compilers may cause a program to abort when attempting to access memory pointed to by two different variables.

Look at two string functions (string1 and string2) that use strlen() and strcopy().

```
char string1[10] = "Merry";
char string2[10] = "Christmas";

cout << string1 << " " << string2 << endl;
cout <<"string1 has length " << strlnen(string1) << " and string2 has length "
     << strlen(string2) << endl;
strcpy(strin2, string1);  // string2 receives a copy of string1
cout << string1 << " " << string2 << endl;
```

`string1` returns the length of a C-string, that is the number of characters it currently holds, not counting the '\0' string terminator character. The `string2` places a copy of one C-string in another.

Output
Merry Christmas
String1 has length 5 an string2 has length 9
Merry Merry

Example program 11.5, which uses a dynamic character array to store the description string, will further illustrate this problem.

Example program 11.5

```
//  This program illustrates problems with memberwise assignment when using pointers

#include <iostream
#include <iomanip>
#include <cstring>
using namespace std;

class Inventory  // class declaration with member functions defined in-line
{
        private:
                char *description   // pointer to the start of an array of characters
                                    //  that will be dynamically allocated
                double price;
        public:
                Inventory()             // default construtor
                {
                        price =0;
                        description = new char[6];
```

```
                strcpy(description, “empty”);
        }
        Inventory (char* d, double p)  // constructor
        {
                description = new char[strlen[d] +1);
                // gets the right amount of memory to hold the string passed in
                strcpy(description, d);
                price = p
        ~Inventory()    // destructor used to free the memory allocated for the
                        // dynamic variable
        {
                delete[] description;
        }

        const char* getDescription()
        {
                return description;
        }

        double getPrice()
        { return price;}

        void setDescription (char* d)     //  “Assumes” dynamic description
        {                       //  variable has enough memory to hold the new string
                strcpy(description, d);
        }

        void setPrice(double p)
        {  price = p;  }
};

int main()
{
        Inventory item1(“hammer”, 12.49);
        Inventory item2 = item1;  // Create item2 and initialize it with item1’s data.

        cout << fixed << setprecision(2) << showpoint;
        cout << “item1: “ << item1.getDescription() << “ $” << item1.getPrice() << endl;
        cout << “item2: “ << item2.getDescription() << “ $” << item2.getPrice() << endl <<endl;

        item1.setDescription(“wrench”);  // Change item1’s description which changes item2’s.

        cout << “item1: “ << item1.getDescription() << “ $” << item1.getPrice() << endl;
        cout << “item2: “ << item2.getDescription() << “ $” << item2.getPrice() << endl <<endl;

        return 0;
```

```
}
```

Output

```
item1:  hammer $12.49                  // item1 was created with this data
item2:  hammer $12.49                  // item2 was created and set = to item1

item1:  wrench $12.49                  // item1's description was changed
item2:  wrench $12.49                  // item2's description was also changed
```

This output illustrates the problem with using memberwise assignment when pointers are involved. Since `item2` was initialized with `item1`'s data, its constructor did not run and so it was never allocated memory to hold its own description. Instead, the address of where item1's description was located was copied into `item2`'s description pointer causing both objects to point to the same memory location. Clearly this is not the result we want. In fact, the problem is considered so sever that some systems report a fatal runtime error when one object attempts to access another object's memory in this manner.

Copy Constructors

Luckily, the use of a copy constructor will solve the above problem in most cases. A copy constructor is a special constructor that is automatically called whenever a new object is created and initialized with another object's data.

A copy constructor looks like a regular constructor except that it must have a reference parameter of the same class type as the object itself. It is through this parameter that it receives the object whose data members are to be copied. The purpose of the copy constructor is to ensure that everything, including pointer values, are transferred to the newly created object in such a way that it has its own copy of everything. Here is a copy constructor for the Inventory class:

```
Inventory(Inventory &object)
{
      description = new char[strlen(object.description) + 1];
      strcpy (description, object.description);
      price = object. Price;
}
```

Notice how this constructor works. It allocates enough memory for the new object to hold the description. Then it copies the description from the object being passed in to the space set up to hold the new object's description. Finally the price, which is a simple variable, not a pointer, is copied from the passed in object to the new object. When this copy constructor is added to the class declaration of Example program 11.5, it will behave properly.

Operator Overloading

Copy constructors solve the problem of memberwise assignment when an object containing pointers is assigned the data values of another object at the time of its creation. However, they don't solve the problem of assigning one object to another at some later time, once the object has already been

created. Constructors, even copy constructors only run when an object is first created. To solve this new problem, we use a technique called **operator overloading**.

Operator overloading allows the programmer to redefine how an existing operator behaves when it is used with a class object. Most C++ operators can be overloaded. To overload an operator in C++ the programmer writes an **operator function**. This is a special public member function that defines how the overloaded operator is to behave. Once this function is included in a class declaration, whenever the operator is used with an object of that class, the operator function is executed to carry out the desired series of operations. To cause the assignment statement to do a correct memberwise copy when one object is "assigned" to another we can overload the assignment operator and write an operator function that looks very muck like the copy constructor.

An operator function is like most other member functions but it has a special name. Its name must be `operator`**`x`** where x is the symbol for the operator to be overloaded.

Example:

```
operator =        //  name of the function to overload the = operator
operator+         //  name of the function to overload the + operator
```

If the operator being overloaded is a binary operator, (i.e. has two operands as = and + do) the operator function must, like a copy constructor, have a reference parameter whose type is the type of the class.

Overloading the Assignment Operator

The following is an operator function to overload the assignment operator for use with objects of the Inventory class.

```
void opearator=(const Inventory &right)
{
      delete [] description;
      description = new char[strlen(right.description) + 1];
      strcpy(description, right.description);
      price = right.price;
}
```

Notice how similar it is to the copy constructor created earlier. It does just one thing that the copy constructor did not do. It includes code to delete dynamic memory previously acquired for the description before it allocates new memory. The copy constructor did not have to do this because it only operates when a new object is created that has no previously acquired dynamic memory. Otherwise the two functions are really the same. The copy constructor called its formal parameter `object` whereas the operator function calls its parameter `right`. This is just a difference of choice. We could have called either formal parameter anything we wanted. When overloading a binary operator, however, it is customary to call the parameter `right` since we thing of the function as operating on the left-hand object and accepting information about the right-hand object through the parameter. In the statement `item3 = item1;` for example, you can think of `item3` as the left operand and `item1` as the right operand of the = operator. Since the assignment operator has been overloaded, this statement invokes `item3`'s `operator=` function, passing it item1 as a reference

parameter. Although it would be unlikely to see it written this way, the same function could be invoked with the following statement:

```
item3.operator=(item1);
```

Operator functions do not have to be void functions. If, for example, we wanted the `operator=` function to return a copy of item3, the object for whom the function is being invoked, we could make the function return type `Inventory` instead of `void`. We would then need to add the following line of code as the last statement in the operator= function.

```
return *this;
```

`this` is a special variable in C++. It is a pointer to the object whose member function if being invoked. In our example, `this` is a pointer to `item3` object. `*this` dereferences the pointer and allows the actual `item3` object to be returned.

The final version of the `operator=` function, with the `Inventory` return type is as follows:

```
Inventory operator=(const Inventory &right)
{
      delete [] description;
      description = new char[strlen(right.description) +1];
      strcpy(description, right.description);
      price = right.price;
      return *this;
}
```

Overloading he Addition Operator

Suppose, in the `Inventory` example, we wanted to keep track of the quantity in stock for each item. We could add the following member variable to the class:

int quantity;

Assume that we have added this member with `item1` having the description “7 inch screwdriver” and the `quantity` 22 while `item2` had the description “9 inch screwdriver” and the `quantity` 14. We might then want to define an overloaded addition operator for the class that would add the quantity values of two objects and return the sum. The `operator+` function for this class would be as follows:

```
int operator+(const Inventory &right)
{
      return quantity + right.quantity;
}
```

It could be invoked as follows:

```
cout <<”Total number of screwdrivers on hand: “ << (item1 + item2);
```

General Operator Overloading Principles

C++ allows most of its operators to be overloaded and, although programmers can overload operators to make them do pretty much anything they want, there are several general principles to keep in mind.

1. It is generally not a good idea to completely change an operator's meaning in a way that is not obvious and reasonable. For example, it would be a bad idea to make the * operator add quantities, rather than multiply them.

2. You cannot change the number of operands taken by an operator. A unary operator, like ++, can have only one operand, the object for which it is called, and thus has no arguments. Binary operators like + or – must have two operands. The left operand is the object for which the operator function is invoked. The right operand should be passed in as a reference parameter.

3. The return type of most operator functions depends on what the overloaded operator is being defined to do rather than which operator is being overloaded. For example, an overloaded addition operator that adds an integer data member from one object to an integer data member from a second object and wants to return their sum would likely have a return type of int. On the other hand, an overloaded addition operator that adds values in a whole set of corresponding data members of two objects might create a temporary object to hold all the sums and then return that entire object.

4. There is one exception to the above general principle. Although an overloaded relational operator (such as == or >) that compares two objects may define in any way it wishes what constitutes being equal or greater than, etc., such a function should always return true or false.

Lab 11: More about Classes and Operator Overloading

The purpose of this lab is to give students experience in working with static member variables and functions, friend functions, copy constructors and overloading operators.

The lab begins with a quiz on the material covered in this lesson.

Fill in the Blank Quiz

1. A(n)_____member variable is shared by all instances of a class.

2. A(n)_____function is a special function that is allowed to be executed before any objects of a class are created.

3. A(n)_____function is a function outside of a class that is allowed to access even private member data and member functions of class objects.

4. A call object_____(can/cannot) be assigned to another object of the same class.

5. In order for a function to be a friend of a class, it must be granted this status by _____.

6. When memberwise assignment occurs_____(all/just the public/just the private) data values are copied from one object to the other object.

7. A(n)_____is used to ensure that memberwise assignment handles pointers correctly when a new object is created and initialized with another object's data.

8. Redefining the way an operator works when applied to objects is called _____.

9. The operator function to redefine the way the subtraction operator works would be named_____.

10. A copy constructor should be passed the object whose values are being copied by _____(value/reference/pointer).

Program11.1 Working with Static Member Variables and Member Functions

Retrieve program `savings.cpp` from the Lab 11 folder. The code is as follows

---

```
// This program introduces the use of static member variables.

// PLACE YOUR NAME HERE
```

```
#include <iostream>
using namespace std;

class SavingsAcct
{
   private:
      int acctNum;                    // "regular" member variables
      double balance;

      // Fill in the code to declare a static int variable named
      // nextAcctNumber.

      // Fill in the code to declare a static double variable named
      // totalDeposits.

  public:
     SavingsAcct();                   // default constructor
     SavingsAcct(double);             // constructor
     void newAcctInfo();
     double getTotalDeposits();
     void displayAcctInfo();
     void deposit(double);
     void withdraw(double);

};

// Static Variable Definitions

// Fill in the code to define the static variable nextAcctNumber and
// initialize it to 5000.

// Fill in the code to define the static variable totalDeposits and initialize
// it to 0.

// Member Function Implementation Section

SavingsAcct::SavingsAcct()               // default constructor

{

     // Fill in the code to assign acctNum the next available account number.
     // Fill in the code to increment the next available account number.



   balance = 0;
```

```
   newAcctInfo();
}

SavingsAcct::SavingsAcct(double startBal)  // constructor

{   // Fill in the code to assign acctNum the next available account number.
   // Fill in the code to increment the next available account number.

   // Fill in the code to set balance to the starting balance passed in.

   // Fill in the code to add this account's starting balance to the static
   // variable named totalDeposits.

   newAcctInfo();
}



void SavingsAcct::newAcctInfo()

{   cout << "New account number: " << acctNum << "   Initial Balance: $"
         << balance << endl;
}


double SavingsAcct::getTotalDeposits()
{   return totalDeposits;
}

void SavingsAcct::displayAcctInfo()
{
  // Fill in the code to display the account's account number and balance.
}



void SavingsAcct::deposit(double amt)
{
  // Fill in the code to implement this function. The amount passed in must
  // be added to the account's balance. Remember that any change to an
  // account's balance also affects totalDeposits.
}

void SavingsAcct::withdraw(double amt)

{   // This function is not being implemented at this time.
```

```
}

// Client code

int main()

{
  // Fill in the code to create acct1 with a starting balance of $100.
  // Fill in the code to create acct2 with a starting balance of $250.

  // Fill in the code to deposit $50 in acct2.

  // Fill in the code to display the account information for acct1.
  // Fill in the code to display the account information for acct2.

  // Fill in the code to display totalDeposits.

  return 0;

}
```

Exercise 1: Complete the program as directed in the bold comments. The following should be the output:

New account number: 5000 Initial Balance: $100
New account number: 5001 Initial Balance: $250
Account number 5000 has a balance of $100
Account number 5001 has a balance of $300
Total savings deposits =$400

Exercise 2: Notice that the `newAcctInfo()` function is never called by any user function outside the class. It is called only by other class functions. This means it can be a `private` member function. Modify the program you completed in Exercise 1 to move the declaration of this member function from the `public` section to the private section and then rerun this program. You should get the same results.

Exercise 3: Modify the program you created in Exercise 2 to add a `static void public` function named `firstAcctNumber` which allows the starting account number to be input to the program , rather than hard coded within it. Write the function prototype and implementation for this function. The static variable `nextAcctNumber` will still need to be defined, as it was before, but it will not be initialized when it is defined. Instead, before creating any accounts, the `main` function should have the user input the starting account number and then call your `fistAcctNumber` function to initialize it with the input number. After making these modifications run your program again, inputting 5000 as the starting account number. The results should be the same as those shown in exercise 1.

Program 11.2 Using Friend Functions

Retrieve salesrep.cpp from the Lab 11 folder. The code is as follows:

```
// This program introduces the use of friend functions.  It also reviews the
// use of static member variables and functions.

// PLACE YOUR NAME HERE.
#include <iostream>
using namespace std;

class SalesForce
{
  private:
    static double salesGoal;       // static variable shared by
                                   // all class objects
    int    snum;                 // sales rep. number
    double monthlySales;

  public:
    static void initGoal(double);  // static function to set salesGoal

    SalesForce(){snum = monthlySales = 0;}  // default constructor
    SalesForce(int, double);       // constructor
    void displayInfo();            // displays an object's member data
    bool metGoal();                // indicates if sales goal has been met

    friend void salesReport(SalesForce &); // declares a friend function
};

// Static Member Variable Definitions

double SalesForce::salesGoal;

// Member Function Implementation Section

void SalesForce::initGoal(double amt)
{
  salesGoal = amt;
}
SalesForce::SalesForce(int num, double amt)   // constructor

{  snum = num;
  monthlySales = amt;
```

```
}

void SalesForce::displayInfo()
{  cout << "Sales Rep Number: " << snum << "  Total sales $" << monthlySales
          << endl;
}


bool SalesForce::metGoal()

{  if(monthlySales >= salesGoal)
     return true;
  else
     return false;
}

// Client Code

// Function prototypes

void salesReport(SalesForce &);

int main()

{
  double goalAmount;
  cout << "Input monthly sales goal: ";
  cin  >> goalAmount;
  SalesForce::initGoal(goalAmount);      // set the sales goal for the class
  SalesForce repOne(101, 13500);         // create 1st sales rep.
  SalesForce repTwo(202, 16800);         // create 2nd sales rep.

  salesReport(repOne);
  salesReport(repTwo);

  return 0;

}


/*************************************************************************

 *                      salesReport                                      *

 * This function produces a sales report using data in a SalesForce      *
```

```
 * object. Since the SalesForce class has declared this a friend function,      *
 * it can access private members of SalesForce class objects directly,          *
 * by using the dot operator, instead of through calls to public member         *
 * functions.                                                                   *
 ******************************************************************************/

void salesReport(SalesForce & rep)
{
  cout << "Report for Sales Rep. number: " << rep.snum << endl;
  cout << "Monthly sales: $" << rep.monthlySales
       << "   Goal: $" << rep.salesGoal << endl;

  if (rep.metGoal())
    cout << "Met sales goal\n\n";
  else
    cout << "Did not meet sales goal\n\n";
}
```

Exercise 1: Run the program inputting 15000 when prompted to enter the monthly sales goal amount. Observe the output. Compare the code to the output to see which statements are creating which outputs.

Exercise 2: Make the following modifications to the program. Add a second `void friend` function to the `SalesForce` class named `modifyGoal` which has two parameters, a type double parameter which receives the amount to raise the goal and a reference parameer that receives the `SAlesForce` object being modified. Write a prototype for this function and the code for Code section. The function is supposed to raise the object's `salesGoal` by the amount passed in. Add a line of code in the `main` function, just before the calls to the `salesReport` function, to call this function to raise `repOne`'s `salesGoal` by 1000.

Exercise 3: Rerun the modified program, again inputting 15000 for the monthly goal amount, an observe th output. Did `repOne`'s `salesGoal` rise to 16000? What happened to `repTwo`'s `salesGoal`?

Explain

Do you fell it is a good idea to allow friend function to modify member data?

Program 11.3 Using Copy Constructors

Bring in program copycon.cpp from the Lab 11 folder. The code is as follows:

```
// This program illustrates the use of copy constructors
// PLACE YOUR NAME HERE.

#include <iostream>
#include <iomanip>
using namespace std;

class Inventory  // class declaration with member functions defined in-line
{
  private
    char   *description;
    double price;

  public:
    Inventory()                       // default constructor
          {  price = 0;

             description = new char[6];
             strcpy(description, "empty");
          }

          Inventory(char* d, double p)            // constructor
          {  description = new char[strlen(d) + 1]; // Get needed amount of memory
                                                    // to hold the description.

             strcpy(description, d);
             price = p;
          }

 ~Inventory()
    {  delete[] description; }         // Use destructor to free the memory
                                       // allocated for the dynamic variable.

    const char* getDescription()
    {  return description;  }



    double getPrice()

    {  return price;  }

    void setDescription(char* d)       // "Assumes" dynamic description
```

```
    {  strcpy(description, d);              // variable has enough memory to hold

    }                                       // the new string.


    void setPrice(double p)
    {   price = p;   }

};

int main()

{  Inventory toolOne("screwdriver", 2.99);

  // Fill in the code to create a new Inventory object named
  // toolTwo that is initialized with the values of toolOne.

  cout << fixed << setprecision(2) << showpoint;
  cout << "toolOne: " << toolOne.getDescription() << "  $"
         << toolOne.getPrice()  << endl;
   cout << "toolTwo: " << toolTwo.getDescription() << "  $"
         << toolTwo.getPrice()  << endl << endl;

  // Fill in the code to change toolTwo's description to "electric screwdriver"

  cout << "toolOne: " << toolOne.getDescription() << "  $"
        << toolOne.getPrice()  << endl;
  cout << "toolTwo: " << toolTwo.getDescription() << "  $"
        << toolTwo.getPrice()  << endl << endl;

 return 0;

}
```

Exercise 1: Fill in the code as indicated in bold type to complete program. Run and observe the output.

Exercise 2: This may cause your program to abort on some systems. If it ran and produced output, rather than aborting, explain why `toolOne`'s description changed.

Exercise 3: Write a copy constructor, and place it in the Inventory declaration with the other constructors. Rerun the program. Now, when `toolTwo`'s description is changed, `toolOne` should not be affected.

Exercise 4: The `setDescription` member function, assumes there has been enough memory allocated for the `description` to hold any new string being passed to it. This is not a safe assumption.

Rewrite the code in this function to free the current memory pointed to by the `description` pointer and allocate the right amount of memory for the new string being passed to the function. Rerun your program with the new function to make sure it still works correctly.

Program 11.4 Overloading Operators

Retrieve program `overload.cpp` from the Lab 11 folder. The code is as follows:

```
// This program contains a revised version of the Inventory class
// and uses it to illustrate operator overloading.

// PLACE YOUR NAME HERE.

#include <iostream>
#include <iomanip>
using namespace std;

class Inventory  // class declaration with member functions defined in-line
                 // a quantity member variable has replaced the price variable

{  private:
     char *description;
     int  quantity;

   public:
     Inventory()                        // default constructor
     {  description = new char[6];
        strcpy(description, "empty");
        quantity = 0;
     }

     Inventory(char* d, int q   )            // constructor
     {  description = new char[strlen(d) + 1]; // Get needed amount of memory
                                               // to hold the description.

        strcpy(description, d);
        quantity = q;
     }



     Inventory(Inventory &object)            // copy constructor
     {  description = new char[strlen(object.description) + 1];
        strcpy(description, object.description);
        quantity = object.quantity;
     }
```

```
 ~Inventory()
 {  delete[] description; }       // Use destructor to free the memory
                                  // allocated for the dynamic variable.

 const char* getDescription()
{  return description;  }

 double getQuantity()
{  return quantity;  }

 void setDescription(char* d)
 {  delete [] description;
    description = new char[strlen(d) + 1];
    strcpy(description, d);
 }

 void setQuantity(int q)
{   quantity = q;   }

 Inventory operator=(const Inventory &right) // Overload the = operator
{
    // Fill in the code to deallocate the memory previously acquired to
    // hold the description.

    // Fill in the code to allocate the right amount of memory to hold the
    // new description.  Place the address of this new memory in the
    // description pointer variable.

    // Fill in the code to set the description of this object to right's
    // description. Remember you need to use strcpy to copy a string.

    // Fill in the code set this object's quantity to right's quantity.

    return *this;

 }



 int operator+(const Inventory &right)      //Overload the + operator
{

    // Fill in the code to return the sum of this object's quantity and
    // right's quantity.
}
```

```
};

int main()
{

  // Fill in the code to create an Inventory object named item1 which is a
  // "7 inch screwdriver" with a quantity of 22.

  // Fill in the code to create an Inventory object named item2 which is a
  // "9 inch screwdriver" with a quantity of 14.

  // Fill in the code to create an Inventory object named item3 which is a
  // "wrench" with a quantity of 10.

  cout << "item1: " << item1.getDescription() << "  "
       << item1.getQuantity()  << endl;
  cout << "item2: " << item2.getDescription() << "  "
       << item2.getQuantity()  << endl;
  cout << "item3: " << item3.getDescription() << "  "
       << item3.getQuantity()  << endl << endl;

  // Fill in the one statement that will assign item3
  // all of the values stored in item2.

  cout << "The following two items should now be identical." << endl;
  cout << "item2: " << item2.getDescription() << "  "
       << item2.getQuantity()  << endl;
  cout << "item3: " << item3.getDescription() << "  "
       << item3.getQuantity()  << endl << endl;

  // Fill in the code to change item3's description back to "wrench".

  cout << "The change to item3 should not affect item2." << endl
        << "item2 should still be a screwdriver." << endl;
  cout << "item2: " << item2.getDescription() << "  "
       << item2.getQuantity()  << endl;
  cout << "item3: " << item3.getDescription() << "  "
        << item3.getQuantity()  << endl << endl;

  cout << "Total number of screwdrivers on hand: "

       << (// Fill in the code to add the quantities of item1 and item2 by
            // using your overloaded + operator, NOT by using getQuantity.)
       << endl;
```

```
  return 0;

}
```

Exercise 1: Fill in the code to complete the program as instruction in the bold type. Run the program. The output should be as follows:

item1: 7 inch screwdriver 22
item2: 9 inch screwdriver 14
item3: wrench 10

The following two items should now be identical.
item2: 9 inch screwdriver 14
item2: 9 inch screwdriver 14

The change to item3 should not affect item2.
item2 should still be a screwdriver.
item2: 9 inch screwdriver 14
item3: wrench 14

Total number of screwdrivers on hand: 36.

Exercise 2. When you changed `item3`'s description it, correctly, did not affect `item2`'s description. Why?

Exercise 3: Write a function to overload the subtraction operator and add it to the class. It should find the difference between two quantities. Test out your overloaded subtraction operator by adding a line of code just before the `return` statement to print out how many more 7 inch screwdrivers (`item1`) there are than 9 inch screwdrivers (`item2`).

Student Generated Code Assignment

Student Generated Code Program 11.5

Write a program that creates a `Circle` class and uses it to find out how far apart the centers of `Circle` objects are. Recall that at the very least `Circle` objects will need to have `radius`, `center_x`, and `center_y` as data members. The class should have a default constructor to set `radius = 1` and to place the center at (0,0). It should also have a constructor to allow all three values to be initialized with user-supplied data when a `Circle` object is created. It should also have functions to `set` and to `get` (i.e., retrieve) each of the data members. Include an overloaded subtraction operator that returns how far apart two `Circle` object centers are. It must not return any negative number. For example if `circle1` is centered at (3,2) and `circle2` is centered at (4,4), the operation `circle1- circle2` should return 2.236, not -2.236. Have the program create a pair of `Circle` objects and then output where their centers are and how far apart they are. Display the result to 3 decimal places.

Sample output (once two circles have been created)

```
circle1 center:  (3,2)
circle2 center:  (4,4)
The two circles are 2.236 units apart.
```

Lesson 11: Summary

Static Member Variables are variables of a class whose value can be shared among all objects created from that class. They are declared by placing the word `static` in front of the declaration in the class declaration section. They are defined by first listing the class name followed by the scope resolution operator and then the name of the variable.

Example declaration

```
class SavingsAcct
{
        private:
                static int nextNumber;
```

Example definition

```
int SavingsAcct::nexNumber = 1001;
```

Static member functions are generally used to initialize static member variables or other setup tasks for a class. A static member function is declared by placing the word `static` in front of its prototype within the calss declaration.

Example declaration

```
class SalesForce
{
   private:
        static double salesGoal;  // static variable
          …..
  public:
        static void initGoal(double);  // static function to set salesGoal
```

Example static member function

```
void   SalesForce::initGoal(double amt)
{  salesGoal = amt:  }
```

Friend Functions are functions that are not members of a class but they have access to private members of a class. The class declaration must declare friend functions. A friend function is declared by preceding its prototype with the word `friend`.

```
class SalesForce
{    private:
        static double salesGoal;  // static variable
          …..
     public:
        static void initGoal(double);  // static function to set salesGoal

        friend void SalesReport(SalesForce &);   // generally placed in public section
```

Memberwise Assignment is a data copy from one object to another.

```
Inventory item3 = item2;  //   This creates a new inventory object and
                          //   initializes its data values with
                         //    another Inventory object's data.
```

Memberwise Assignment can create problems when pointers to dynamic memory are involved.

A **copy constructor** is a special constructor that is automatically called whenever a new object is created and initialized with another object's data. It has a reference parameter of the same class type as the object itself. It is through this parameter that it receives the object whose data members are to be copied. In this way the object has its own copy of everything.

```
Inventory(Inventory &object)
{
      description = new char[strlen(object.description) + 1];
      strcpy (description, object.description);
      price = object. Price;
}
```

Operator overloading is the process of redefining how an existing operator behaves when it is used with a class object. To overload an operator, the programmer writes an **operator function**. All operator functions have the name `operatorX`. Where X is the operator.

Example

```
operator =        //  name of the function to overload the = operator
operator+         //  name of the function to overload the + operator
```

If the operator being overloaded is a binary operator, (i.e. has two operands as = and + do) the operator function must, like a copy constructor, have a reference parameter whose type is the type of the class.

The following is an operator function to overload the assignment operator for use with objects of the Inventory class.

```
void opearator=(const Inventory &right)
{
      delete [] description;
      description = new char[strlen(right.description) + 1];
      strcpy(description, right.description);
      price = right.price;
}
```

Lesson 12: Header Files, Inheritance & Recursion

Abstraction

Abstraction encompasses the idea that implementation of complex programming logic does not have to be known to the user in order to be useful. For example a person does not have to know how a remote controller works in changing channels in order to use the remote. The user does need to know however which buttons to push to accomplish the various features that are offered by the remote. It is the same in programming. Complex programs are written on the back of other complex programs whose implementation logic is hidden. Object Oriented programmers often use objects derived from pre-existing classes written by other programmers. It is important for programmers to understand how to use the class functions (methods) to accomplish a certain task, just as the remote user needs to know which buttons to push, but it is not necessary for the programmer to know how those functions are implemented just as the remote user does not have to understand how a remote works.

For that reason class declarations and the implementations of their functions are often saved in special files.

Header Files

Header files are used in C++ to allow programmers to separate certain elements of a program (such as class declarations and function implementations) into reusable files so that many programmers can use them without the need to understand the underlying logic. Header files may contain variables, procedures, declarations of classes and other identifiers.

Header files are named with a .h suffix. They are included in a program by using a compiler directive.

Compiler directive

They symbol # before an instruction indicates that it is a compiler directive. It is a symbol that gives a command to the compiler to do something rather than a fundamental instruction of the program. The most common directive used is the include directive which tells the compiler to include as part of the program whatever is enclosed in <> which follows the command.

We have been using this command throughout the course:

`#include <iostream>` is a complier directive to include a standard ANSI (American National Standards Institute) header file that includes, among other things, the `cin` and `cout` definitions so that the user can use this short hand method of inputting and outputting data to and from a program. The ANSI have many header files that do not have the .h extension but user created header files will.
We must make sure that a header file is not included more than once. For that reason all header files should include the following compiler directives:

```
#ifndef     //  This is a conditional directive that means if not defined
#define     //   This is a directive to define this file
#endif      //    This is the end of the conditional directive
```

Generally speaking header files should follow the following format:

```
#ifndef    NAME_H
#define    NAME_H

     contents of file



#endif
```

The inclusion of these directives prevents a header file from accidentally being included more than once. The `#ifndef` directive checks for the existence of a manifest constant NAME_H. If the constant has not been defined, it is immediately defined and the file is included. If the constant has been defined, everything between the `#ifndef` and `#endif` directives is skipped.

Class declarations (the list of private and public members) and the class member functions implementation code should be separated into separate files. That means that the file containing the function implementation code should include the header file containing the class declaration.

Components of a program are generally stored as follows:

Class declarations are stored in their own header files. A header file that contains a class declaration is called a **class specification file**. The name of the class specification file is usually the same as the class with a .h extension.

Member function definitions are stored in a separate .cpp file called the **class implementation file.** The file usually has the same name as the class with the .cpp extension.

Any program that uses the class should #include the class's header file. The class's .cpp file (that which contains the member function definitions) should be compiled and linked with the main program. This process can be done automatically by the project in the IDE or by a make utility.

Although the ANSI header files use the <> symbols, user defined header files are included through the use of double quotes.

These concepts are best understood by examples.

The following three files include a header file that declares a class called `Rectangle`, the `.cpp` file that implements the member functions and a client `.cpp` code that use the `Rectangle` class.

Example program 12.1 Rectangle class header file The file is `rectangle.h`

```
#ifndef  RECTANGLE_H
#define RECTANGLE_H

// Declaration of  Rectangle class

class Rectangle
{
        private:
                float width;
                float length;
                float area;
        public:
                void setData(float, float);
                void calcArea();
                float getWidth();
                float getLength((;
                float getArea();
};
#endif
```

Example program 12.2 Rectangle member function implementation.
The file is `rectangle.cpp`

```
#include <iostream>
#include "rectangle.h"    // NOTE THE USE OF DOUBLE QUOTES
                          // NOTE that this file must include the header file

using namespace std;

//***************************************
//                    setData
// setData copies the argument w to private
// member width and l to private member length
//***************************************
void Rectangle::setData(float w, float l)
{
        width =w;
        length = l;
}

//***************************************
//                  calcData
//  calcData finds area and places result in
//  private member area
//***************************************
```

```
void Rectangle::calcArea()
{
        area = width * length;
}


//************************************
//    getWidth and getLength  and getWidth
//    returns the private members width
//    length and area respectively
//************************************

float Rectangle::getWidth()
{   return width; }

float Rectangle::getLength()
{    return length; }

float Rectangle::getArea()
{    return area;   }
```

Example program 12.2 client file example file that uses the Rectangle class The name can be anything.cpp

---

```
#include <iostream>
#include "rectangle.h"   // must include the header file.  The implementation file would
                         //  be compiled along with this client file
                         //  In using an IDE project all three files would be included in the
                        //   project.
using namespace std;

int main()
{
        Rectangle box;
        float boxWidth, boxLength;
        cout << "Please input the width of the box" << endl;
        cin >> boxWidth;
        cout << "Please input the length of the box" << endl;
        cin >> boxLength;
        box.setData(boxWidth, boxLength);
        box.calcArea();
        cout << "Box Width:  " << box.getWidth() << endl;
        cout << "Box Length:  " << box.getLength() << endl;
```

```
        cout << "Box Area:  " << box.getArea() << endl;
        return 0;
}
```

From this point on you should follow this format for writing your programs.

Inheritance

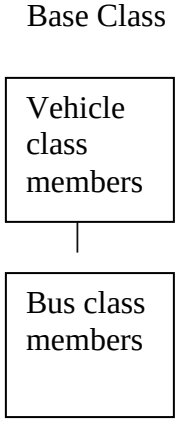
Inheritance, in object oriented programming, is the idea that a class can be extended to include other items such as functions etc. to create a new class. The new class "inherits" all the members of the original class. It is based on the concept of a sub-class. A sub-class can be described as a specialized version of a larger class. For example: beagle is a sub-class of dog. Everything that describes a dog can be used to describe a beagle; however a beagle has certain characteristics that may not be true of other types of dogs. This describes what is called an "is a" relationship. A beagle *`is a`* dog. The following is a list of common "is a" relationships:

A woman `is a` person
A bus `is a` vehicle
A salmon `is a` fish
A rose `is a` flower

Notice that the noun to the left of the "is a" statement is a component of the noun to the right of that statement. All busses are vehicles. The inverse is not true. Not all vehicles are busses. The specialized object (the noun to the left of the "is a statement) has all of the characteristics of the general object (the noun to the right of the "is a" statement), plus additional characteristics that make it special.

Inheritance is used in object-oriented programming to create an "is a" relationship between classes. Inheritance describes a base class (the general class or noun to the right of the "is a" statement) and a derived class which is the specialized class.

The derived class "inherits" the members (both member variables and member functions) of the base class without any of them rewritten. The derived class can then have extra "specialized" members that are not part of the base class.



Derived Class

Example:

Suppose we want to make a distinction between show dogs and other types of dogs. A show dog "is a" dog. Thus dog is our base class and show do is our derived class. For our purpose, let us assume that all dogs have a color, weight and name. These are the private member variables of our base class. In addition to these characteristics, let us assume that show dogs have the following characteristics:

Number of contests entered: `numOfContests`
Number of first place finishes: `numOfWins`

The private member variables of a show dog will also include these items.

The base class can be declared as follows:

```
class DogType                                   // name of the base class
{
        private:
                string name;                    //  name of dog
                float weight;                   // weight of dog
                string color;                   // color of dog
        public:
                DogType();                      //default constructor
                DogType(string n, float w, string c);   //  constructor
                void printDogInfo() const;      // prints dog information
};
```

The following is the implementation of the member functions

```
DogType::DogType()                              // default constructor
{
        name = "Unknown";
        weight = 0;
        color =  "Unknown";
}


DogType::DogType(string n, float w, string c)   // construtor
{
        name = n;
        weight = w;
        color = c;
```

```
}

void DogType::printDogInfo()  const
{
        cout << "The dog's name is " << name << endl;
        cout << "The dog's weight is " << weight << endl;
        cout << "The dog's color is " << color << endl;
}
```

The derived class `ShowDogType` will inherit the members of `Dogtype`. It's declaration is as follows:

```
class ShowDogType: public DogType
        // name of the derived class followed by a colon, the word public to indicate
        // that it is inheriting members, and the name of the base class from which
        // it inherits those members
{
        private:
                int numOfContests;                  //  number of contests entered
                int numOfWins:                      //  number of contests won
        public:
                ShowDogType(string n, float w, string c, int con, int win);
                //  This is a constructor that will give values to all the member
                //   variables of this class including the ones it inherits.

                ShowDogType();                      //  default constructor
                void printShowDogInfo() const:      //  prints contests & wins
};
```

Although this derived class (`ShowDogType`) inherits the members of its base class both private and public, it can access only the public members of the base class. To change the private members of `DogType, ShowDogtype` must access the public members just as other users would have to do.

The following is the implementation of `ShowDogType` member functions.

```
ShowDogType::ShowDogType(stirng n, float w, string c, int con, int win) :  DogType(n,w,c)

{
        numOfContests = con;        // assigns con to private variable numOfContests
        numOfWins = win:            // assigns win to private variable numOfWins
}
```

Notice that this constructor calls the DogType constructor to initialize the private member variables defined in DogType. Since name, weight, and color are defined in the base class, the derived class cannot access them and thus must call base class constructor. It does this by inserting a colon just before the call.

```
ShowDogType::ShowDogType():  DogType()
{
        numOfConstests =0;          // set default contests to 0
        numOfWins = 0;              // set default wins to 0
}

//  The other member variables (name, weight, color) are given default values
//  by the call to the default constructor DogType()

void ShowDogType::printShowDogInfo() const
{
        printDogInfo();
        cout << “The dog has been entered in ” << numOfContests << “ contests”
             << endl;
        cout << “The dog has won ” << numOfWins << “ contests” << endl;
}
```

The following is a complete set of programs (header files, implementation files and client file) that uses inheritance.

Example Program 12.4a DOGTYPE header file DogType.h

```
#include <iostream>
#include <string>
using namespace std;
#ifndef  DOG_H
#define DOG_H



class DogType                                          // name of the base class
{
      private:
            string name;                               //  name of dog
            float weight;                              // weight of dog
            string color;                              // color of dog
      public:
            DogType();                               //default constructor
DogType(string n, float w, string c);               //  constructor
            void printDogInfo() const;              // prints dog information

};
#endif
```

Example program 12.4 b implementation of DogType class Dogtype.cpp

```
#include<iostream>
#include "DogType.h"
using namespace std;

DogType::DogType()                                      // default constructor
```

```
{
	name = "Unknown";
	weight = 0;
	color =  "Unknown";
}


DogType::DogType(string n, float w, string c)		// construtor
{
	name = n;
	weight = w;
	color = c;
}

void DogType::printDogInfo()  const
{
	cout << "The dog's name is " << name << endl;
	cout << "The dog's weight is " << weight << endl;
	cout << "The dog's color is " << color << endl;
}
```

Example program 12.4 c ShowDogType header file ShowDogType.h

```
#include<iostream>
#include "DogType.h"
using namespace std;
#ifndef  SHOWDOG_H
#define  SHOWDOG_H
class ShowDogType: public DogType
	// name of the derived class followed by a colon, the word public to indicate
	// that it is inheriting members, and the name of the base class from which
	// it inherits those members
{
	private:
		int numOfContests;		//  number of contests entered
		int numOfWins;			//  number of contests won
	public:
		ShowDogType(string n, float w, string c, int con, int win);
		//  This is a constructor that will give values to all the member
		//   variables of this class including the ones it inherits.

		ShowDogType();			//  default constructor
		void printShowDogInfo() const;	//  prints contests & wins
};
#endif
```

Example program 12.4d ShowDogType implementation file ShowDogType.cpp

```
#include <iostream>
#include "DogType.h"
#include "ShowDogType.h"
#include <string>

using namespace std;
```

```
ShowDogType::ShowDogType(string n, float w, string c, int con, int win) :
DogType(n,w,c)

{
      numOfContests = con;    // assigns con to private variable numOfContests
      numOfWins = win;        // assigns win to private variable numOfWins
}

ShowDogType::ShowDogType():  DogType()
{
      numOfContests =0;       // set default contests to 0
      numOfWins = 0;          // set default wins to 0
}

//  The other member variables (name, weight, color) are given default values
//  by the call to the default constructor DogType()

void ShowDogType::printShowDogInfo() const
{
      printDogInfo();
      cout << "The dog has been entered in " << numOfContests << " contests"
                  << endl;
      cout << "The dog has won " << numOfWins << " contests" << endl;
}
```

Example program 12.4e client file

```
#include<iostream>
#include "ShowDogType.h"

using namespace std;

int main()
{
      ShowDogType dog1 ("Fido", 12, "Black", 12,2);
      ShowDogType dog2;

      dog1.printShowDogInfo();
      dog2.printShowDogInfo();

      return 0;
}
```

Note: In the client file it was not necessary to include the DogType.h file. Why?

Running the program produces the following output:

The dog's name is Fido
The dog's weight is 12
The dog's color is Black
The dog has been entered in 12 contests
The dog has won 2 contests
The dog's name is Unknown
The dog's weight is 0

The dog's color is Unknown
The dog has been entered in 0 contests
The dog has won 0 contests

Recursion

We know that a function (method) can call another function (method). But what would happen if a function called itself? Such a function is called a **recursive function**. The problem with a recursive function is that if there is not some condition to cause it to stop calling itself, it will act like an infinite loop. It must have some instruction to control the number of times it repeats itself. Recursion must then be done with great care. Nevertheless recursive functions are very useful in solving certain problems that have a repetition feature inherit in their solution. The art and skill of developing recursive functions has led the old saying "To err is human, to forgive divine" to be rewritten to "To reiterate is human, to recursive divine."

We first look at at example that although done recursively is better done through loops (reiteration). This program has a function that will print a message. It calls itself and thus prints the message again. The number of calls is controlled by a parameter given by the main program. Both the main and the function is given below.

Example Program 12.5 a simple recursive function to repeat a message a certain amout of times

```
#include <iostream>
using namespace std;

void message(int);    // function prototype

int main()
{
      message(3);
      return 0;
}

// ****************************************************************
//                                                  message function
// The control of this recursive function is:  if the parameter is greater than 0,
the
// the message is displayed and the function is recursively called with an
argument 1
// less then the previous call: times -1
// data in:    parameter to indicate the numer of calls to the function itself
// data out:  a message each time
//*****************************************************************

void message (int times)
{
      cout <<  "This message is called with the number " << times << "  as the
parameter.\n";
      if (times >0)
      {
            cout << "I love you"<< endl;
            message(times-1);
      }
```

```
      cout << "This message is now returning with the number " << times
                << " as the parameter \n";
}
```

The `cout << "This message is now returning with the number " << times << " as the parameter \n";` was included just to show the mechanism of a recursive function. Recursive functions go a certain depth in the call and then backtrack back up the path.

The program will produce the following output.

This message is called with the number 3 as the parameter.
I love you
This message is called with the number 2 as the parameter.
I love you
This message is called with the number 1 as the parameter.
I love you
This message is called with the number 0 as the parameter.
This message is now returning with the number 0 as the parameter
This message is now returning with the number 1 as the parameter
This message is now returning with the number 2 as the parameter
This message is now returning with the number 3 as the parameter

Recursion breaks down complex taxks into a solvable problem. The solvable problem is called the **base case**. A recursive function is designed to terminate when it reaches its base case.

Let's look at a problem that is suited for recursion.

The Fibonacci sequence, named after the Italian Leonardo Fibonacci circa 1170 is as follows:

0, 1,1,2,3,5,6,13,21,34,55,89,144,233, 377……

After the second number, each number in the series is the sum of the two previous numbers.

Since this pattern repeats itself it is a good candiate for recursion.

The following is an algorithm for this series that uses recursion:
Given some value n where n is the nth-1 element of the fibonacci sequence)

```
int fib(int n)
{
  if  (n <=0)
     return 0;
 else if ( n==1)
    return 1
  else
     return fib(n-1) + fib(n-2)
}
```

The following program represents this sequence in C++ code.

Example program 12.6 Fibonacci Sequence

```
#include <iostream>
using namespace std;

int fib(int);    // function prototype

int main()
{
      cout <<"The first 15 Fibonacci numbers are:\n";
      for (int x=0; x < 15; x++)
                  cout << fib(x) << " "<< endl;
      return 0;


}

// ****************************************************************
//                           Fibonacci Function
// This function retrns the nth Fibonacci number
//****************************************************************

int fib (int n)
{
      if (n<=0)
            return 0;
      else if (n==1)
            return 1;
      else
            return (fib(n-1) + fib(n-2));
}
```

Notice that the fibonacci function calls itself twice! The first time to find the number 1 less then n and the second time to fine the number 2 less than n. These two numbers are added together to give the correct number in the sequence.

The factorial operation is another problem that is solved *divinely* by recursion. That solution will be left as a lab exercise.

Lab 12: Header Files, Inheritance & Recursion

The purpose of this lab is to give students experience in creating header files and working with the concepts of inheritance and recursion

The lab begins with a quiz on the material covered in this lesson.

Fill in the Blank Quiz

1) _____is based on the idea that the implementation of complex programming logic does not have to be known to the user in order to be useful.

2) _____files allow programmers to separate elements such as class declarations and function implementations into reusable files.

3) The _____ is the conditional directive that means “if not defined”.

4) _____is the idea that a class can be extended to include other items inorder to create a new class.

5) The _____class “inherits” the members of the _____class.

6) Inheritance is used in object-oriented programming to create a(n) _____ relationship.

7) A function that calls itself is a(n) _____function.

8) The solvable problem of a recursive task is called the _____ case.

9) The sequence of numbers 0,1,1,2,3,5,6,13,21……is called the _____sequence.

10) A header file that contains a class declaration is called a(n) _____file.

Program12.1 Creating header files

Retrieve program `circleStats.cpp` from the Lab 12 folder. The code is as follows

---

```
#include<iostream>
#include"CircleClass.h"
using namespace std;


int main()
{
      Circle shape;
      double shapeRadius;
```

```
      cout << "Please input the radius of the circle"<<endl;
      cin >> shapeRadius;
      shape.setRad(shapeRadius);
      shape.calcArea();
      shape.calcCircumference();
      cout <<"These are the stats for a circle with a radius of "
           <<shape.getRadius()<<endl;
      cout <<"The area of the circle is " << shape.getArea()<< endl;
      cout <<"The circumference of the circle is "
           << shape.getCircumference()<<endl;

      return 0;
}
```

Exercise 1 This client program includes a header file for the `Circle` class. You must create a project that includes this client `.cpp` program. You then should make a header file called `CircleClass.h` that has the `Circle` class declaration and a `.cpp` program called `CircleClass.cpp` that has the function implementations of the class. The Circle class should have the following members. NOTE: use 3.141593 for PI.

```
private:
      double radius;
      double area;
      double circumference;
public:
      void setRad(double);
      void calcArea();              // finds the area of the circle using private
                                    // member data radius
      void calcCircumference();     // finds circumference using radius
      double getRadius();           // returns the radius
      double getArea();             // finds and returns area of circle
                                    // using private member radius
      double getCircumference();    //  finds and rerturns circumference
```

Exercise 2: Run the program to make sure it works.

Sample Run:
Please input the radius of the circle
5
These are the stats for a circle with a radius of 5
The area of the circle is 78.5398
The circumference of the circle is 31.4159

Program 12.2 Inheritance

Retrieve program `inherit.cpp` from the Lab 12 folder. The code is as follows:

```
// This program demonstrates the use of inheritance
// PLACE YOUR NAME HERE

#include <iostream>
#include <string>
using namespace std;
```

```
class DogType                // name of the base class
{
    private:
        string name;    // name of dog
        float weight;   // weight of dog
        string color;   // color of dog
    public:
        DogType()
        {
                name = "Unknown";
               weight = 0;
          color = "unknown";
        } // default constructor

        DogType(string n, float w, string c)
        {
                name = n;
                weight = w;
                        color = c;
        }  // constructor

        void printDogInfo() const   // prints dog information
        {
                cout << "The dog's name is " << name << endl;
                cout << "The dog's weight is " << weight << endl;
                cout << "The dog's color is " << color << endl << endl;
        }
    };

class ShowDogType: public DogType
    // The name of the derived class followed by a colon, then the word
    // public to indicate that it is inheriting members, and finally the
    // name of the base class from which it inherits those members
{
      private:
            int numOfContests;  // number of contests entered
            int numOfWins;      // number of contests won
      public:
            ShowDogType::ShowDogType(string n, float w, string c, int con,  int win): DogType( n, w, c)
           {
                numOfContests = con; // assigns con to private variable numOfContests
                numOfWins = win;     // assigns win to private variable numOfWins
            }

           ShowDogType::ShowDogType(): DogType()
           {

                numOfContests = 0;  // set default contests to 0;
```

```
            numOfWins = 0;      // set default wins to 0;
        }

        // The other member variables (name, weight, color) are given
        // default values by the call to the default constructor DogType()

        void ShowDogType::printShowDogInfo() const
        {
            printDogInfo();
            cout << "The dog has been entered in " << numOfContests
                 << " contests" << endl;
            cout << "The dog has won " << numOfWins << " contests" << endl <<endl;
        }
    };

//  Fill in the code to create the class RaceDogType.  See Exercise 1 below

int main()
{
   ShowDogType  dog1("Fido", 12, "Black", 12, 2);
   ShowDogType  dog2;

// Fill in the code to define an object called dog3 that has the following information:
// name --->  Anchovi
// weight --> 14
// color ---> Black
// number of races -----> 28
// number of wins  -----> 27

// Fill in the code to define an object calle dog4 that has all default values


dog1.printShowDogInfo();
dog2.printShowDogInfo();

// Fill in the code to print out all the information about dog3
// Fill in the code to print out all the information about dog4

   return 0;
}
```

Exercise 1: Create another class called RaceDogType that is derived from the DogType class. It willinherit all the members from DogType. It will also have the additional following members:

Private member variables:

1) numOfRaces // an integer value that keeps track of the number of races the dog entered.
2) numOfRacesWon // an integer value that keeps track of the number of races the dog won

Public member functions and constructors

1) Default constructor that sets numOfRaces and numOfRaces Won to 0. It should call the default constructor of DogType to make sure the inherited variables are also initialized.
2) Constructor that will take values for ALL private member variables (those inherited as well as those listed in the derived class). Make sure you correctly call the appropriate constructor in the base class
3) A member function (void printRaceDogInfo() const) that will print ALL the information about a race dog.

Fill in all the code (in bold) so that the following output is produced when run.

The dog's name is Fido
The dog's weight is 12
The dog's color is Black

The dog has been entered in 12 contests
The dog has won 2 contests

The dog's name is Unknown
The dog's weight is 0
The dog's color is unknown

The dog has been entered in 0 contests
The dog has won 0 contests

The dog's name is Anchovi
The dog's weight is 14
The dog's color is Black

The dog has been entered in 28 races
The dog has won 27 races

The dog's name is Unknown
The dog's weight is 0
The dog's color is unknown

The dog has been entered in 0 races
The dog has won 0 races

Exercise 2: Using the code created in Exercise 1 create a new project that will have the following files:

`dogType.h` // a header file that has the class DogType declaration
`dogType.cpp` // a .cpp file that has the DogType member function implementations
`raceDogType.h` // a header file that has the RaceDogType declaration (it inherits the dogType class)
`raceDogType.cpp` // a .cpp file that has the RaceDogType member function implementations
`dogShow.cpp` // a Client file that uses the RaceDogType class

NOTE: You do not have to include the `ShowDogClass` in your code. The `dogShow` client file will just create two instances of the `RaceDogType` `dog1 and dog2`. `dog1` has the following information
// name ---> Anchovi
// weight --> 14
// color ---> Black
// number of races -----> 28
// number of wins -----> 27
dog2 has the default values.

The `dogShow.cpp` file produces the following output

The dog's name is Anchovi
The dog's weight is 14
The dog's color is Black

The dog has been entered in 28 races
The dog has won 27 races

The dog's name is Unknown
The dog's weight is 0
The dog's color is unknown

The dog has been entered in 0 races
The dog has won 0 races

Program 12.3 Recursion Finding greatest common denominator

The greatest common divisor of two positive integers can be found by using Euclid's Algorithm which states that: Given two positive integers x and y the gcd(x,y) is
if y divedes x evenly
gcd is y
else
gcd (y, remainder of x/y)

Note: remember that the operation % gives the integer remainder of two integers.
Bring in problem `gcd.cpp` from the Lab 12 folder. The code is as follows:

```
#include <iostream>
using namespace std;

// This program uses a recursive function to calculate the
// greatest common divisor (gcd) of two integers using Euclid's
// algorithm

// Function prototype
int gcd(int,int);

int main()
```

```
{
        int firstNumber, secondNumber;
        cout << "This program will find the greatest common divisor of two numbers"<<endl;
        cout <<"Please input the first positive number now" << endl;
        cin >> firstNumber;
        cout << "Please input the second postivie number now" << endl;
        cin >> secondNumber;
        cout <<"The greates common divisor of " << firstNumber << " and "
                << secondNumber << " is ";
        cout << gcd(firstNumber, secondNumber) << endl;

        return 0;
}

//*****************************************************************
//                      gcd
//  This function finds the greatest common divisor of two
//  numbers by using Euclid's algorithm recursively
//
//  data in:  num1  num2   two integers
//  data out:  the greatest common divisor of the two numbers
//
//*****************************************************************

int gcd(int num1, int num2)
{
        //  Fill in the if else condition and instructions that will
        //  find the greatest common divisor through the use of
        //  recursion

}
```

Exercise 1: Fill in the code (the part in bold) to complet the gcd recursive function.
Exercise 2: Run the program several times with different numbers
Example run

```
This program will find the greatest common divisor of two numbers
Please input the first positive number now
100
Please input the second postivie number now
52
The greates common divisor of 100 and 52 is 4
```

Student Generated Code Assignments

Student Generated Code Program 12.4

Write programs (including header files and member function implementations) that creates a `Car` class that includes private members that identify the car's `model` (ex. Chevy), `year` (ex. 2010),and `miles` driven (ex. 30000). The public members consist of a constructor that has values for each private member data and a default constructor that sets model to "unknown", year to 0000 and miles to 0. It has a member function that asks the user for the current year and then determines the average number of miles the car has been driven in a year. It also has a member function that prints the car's information (model, year and miles driven).

Create a `RaceCar` class that inherits the `Car` class but includes the number of races the car entered and the number of races won. The `RaceCar` class should include a constructor that has values for number of races entered and number of races won and a default constructor that sets both of those to 0. The `RaceCar` class should also include a member function that determines the percentage of wins the car has. It also has a member function that prints the `RaceCar`'s information: all the information from the `Car` class plus the number of races entered and number of races won and the percentage of the races won.

This project must include 2 header files (one for the `Car` and one for the `RaceCar` declaration), 2 member function implementations (one for the `Car` and one for the `RaceCar` class) and a client file. You may use the following client file:

```
#include<iostream>
#include<string>
#include"RaceCar.h"
using namespace std;

int main()
{
	Car auto1("Dodge", 2000,60000);
	Car auto2("Chevy", 2009,20000);
	Car auto3;
	auto1.printCarInfo();
	auto1.averageMiles();
	auto2.printCarInfo();
	auto2.averageMiles();
	auto3.printCarInfo();
	auto3.averageMiles();
	RaceCar racer1("Dodge", 2008, 70000, 10, 3);
	RaceCar racer2;
	RaceCar racer3("Ford", 2005, 125000, 7,1);
	racer1.printRaceCarInfo();
	racer2.printRaceCarInfo();
	racer3.printRaceCarInfo();
	return 0;
}
```

A sample run of the program will produce the following:

The model of the car is Dodge
The year of the car is 2000
The car has been driven 60000 miles

Please input current year
2010
The car has been driven an average of 6000 miles per year

The model of the car is Chevy
The year of the car is 2009
The car has been driven 20000 miles

Please input current year
2009
You just bought the car this year

The model of the car is unknown
The year of the car is 0
The car has been driven 0 miles

Please input current year
2010
The car has been driven an average of 0 miles per year

The model of the car is Dodge
The year of the car is 2008
The car has been driven 70000 miles

The car has been run in 10 races.
The car has won 3 races.
The car has won 30% of the races

The model of the car is unknown
The year of the car is 0
The car has been driven 0 miles

The car has been run in 0 races.
The car has won 0 races.
The car has won 0% of the races

The model of the car is Ford
The year of the car is 2005
The car has been driven 125000 miles

The car has been run in 7 races.
The car has won 1 races.
The car has won 14.2857% of the races

Student Generated Code Program 12.5

Write a program that includes a function that will find the factorial of a number entered by the user.
Note: 4 ! (four factorial) = 4 * 3*2*1 = 24
The factorial of 0 is 1
0!=1 1!=1 2!=2 3!=6 etc.
Note: Factorials grow very large. In your samle runs do not use very large numbers.

Sample Run:

Please enter a number and the program will return its factorail
9
9! = 362880

Lesson 12: Summary

Header files are used in C++ to allow programmers to separate certain elements of a program (such as class declarations and function implementations) into reusable files so that many programmers can use them without the need to understand the underlying logic. Header files may contain variables, procedures, declarations of classes and other identifiers. Header files are named with a .h suffix. Header files should have the following format:

```
#ifndef    NAME_H
#define    NAME_H
     contents of file
#endif
```

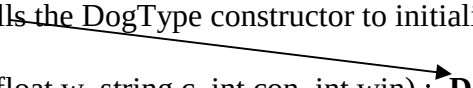
Header files generally contain a class declaration. A separate .cpp file (with the same name as the header file except it has a .cpp extension instead of .h) should be written for the class member function implementation.

Inheritance

Inheritance is used in object-oriented programming to create an "is a" relationship between classes. Inheritance describes a base class (the general class or noun to the right of the "is a" statement) and a derived class which is the specialized class.

The derived class "inherits" the members (both member variables and member functions) of the base class without any of them rewritten. The derived class can then have extra "specialized" members that are not part of the base class.

The following is an example of a derived class constructor that uses the constructor of the base class. The base class (DogType) has private members name(stirng), weight(float), and color(string). The derived class has numOfContests and numOfWins as private members, but it "inherits" the member functions of DogType. This statement calls the DogType constructor to initialize its member data.



```
ShowDogType::ShowDogType(stirng n, float w, string c, int con, int win) : DogType(n,w,c)

{
        numOfContests = con;        // assigns con to private variable numOfContests
        numOfWins = win:            // assigns win to private variable numOfWins
}
```

Recursion

A **recursive function** calls itself. The problem with a recursive function is that if there is not some condition to cause it to stop calling itself, it will act like an infinite loop. It must have some instruction to control the number of times it repeats itself. Nevertheless it is very useful in many applications such as finding factorial (where a process is repeated until some condition is met).

Lesson 13 Namespaces, Event Handling Routines & Enumerated Data

We have been using the following directive (command) in all our programs:

using namespace std;

What does it mean and why are we using it?

Namespaces deal with the concept of scope. You recall that scope defines areas in the program where identifiers (variables, constants etc.) are active. Namespaces allow the programmer to create a scope for global variables. This is necessary when many programmers or multiple software packages are involved. More than one programmer, or software package, may use the same name for a variable thus creating a conflict of names. This problem can be solved if each programmer, or software vendor, places each set of its classes, and identifiers in a separate place called a namespace that has its own unique name.

Every programming language has a philosophy with the way the names of variables are managed. A common problem in large programs is that different modules both want to use the same name at global scope for different things. Namespaces are used to solve this problem. The "namespace" syntax is used to restrict the scope of a collection of names in order to avoid name clashes with other identifiers used elsewhere in the program. Every object or type defined in a namespace has both a "first name" (e.g "cout") and a last name (e.g., "std").

An object defined outside of every namespace is said to be in the "global namespace" and has no "last name." It has become standard practice to avoid "polluting" the global namespace as much as possible; i.e., keep it empty except for namespaces and perhaps some functions or classes whose types are known to all other components of the program.

When using an identifier, the programmer must indicate the name of the namespace. This can be done by using the scope resolution operator ::

Defining a Namespace

A namespace is defined as follows:

```
namespace namespace_name
{
        Declarations
}
```

The following gives an example of namespace definition and use:

```
namespace dino                          int main ()
{                                       {
```

```
    int count=1;
    bool addo()
    {
        return (count%2)
    }
    const int N = 30;
}
```

```
 for (int i=0; i < dino::N, i++)
    {
        dino::count++;
            if (dino::addo() )
                std::count << dino::count << std::endl;
     }
 return 0;
}
```

Here “count” is somewhat like a global variable, but safer, because it is wrapped inside the “dino” namespace. Hence, accidental name clashes with it are unlikely.

You can use something defined in the namespace in code outside the namespace, but to do so, you must specify it uniquely by one of the following 3 techniques.

1) Refer to it by its full name. eg. std::cout << “hello” << std::endl;
   This uses the scope resolution operator ::
2) Instruct the compiler to automatically use the namespace version of the object in the current context: eg.

```
using std::cout;    // you make this “using declaration” only once

cout << “hello” << std::endl;  // I still have to use std:: in front of endl.  Why?
```

   This technique is also safe, but a separate declaration is required for each identifier from the namespace.

3) Instruct the compiler to introduce “all” identifiers in the relevant namespace.

```
e.g.        using namespace std:

            cout << “hello” << endl;
```

Review of Header files & Compiler Directives

Header files are used in C++ to allow programmers to separate certain elements of a program into reusable files. Header files may contain variables, procedures, declarations of classes and other identifiers.

The symbol # before an instruction indicates that this is a compiler directive. That means it is a command to the compiler to do something rather than a fundamental instruction of the program. The most common directive used is the include directive which tells the compiler to include as part of the program whatever is enclosed in < > which follows the command.

In essence there is what might be described as a second language of a program that speaks directly to the compiler.

iostream header file

The ANSI (American National Standards Institute) C++ standard offers a header file called `iostream` which includes a variety of commonly used ANSI standard identifiers such as string, vector, queue, list, cout, cin, endl etc. all of which are part of the namespace called std.

The iostream header file includes the namespace std as which contains as we noted many useful identifiers.

That is why we included the following in all our programs:

This is why all our programs included the following:

```
#include <iostream>
using namespace std;
```

We want access to this header file called `iostream` and we want to access all the identifiers in the std namespace without having to use the scope resolution operator before each of those identifiers. That is why we used the third method given above for our programs.

Instruct the compiler to introduce "all" identifiers in the relevant namespace.
e.g. using namespace std:

cout << "hello" << endl;

This technique is the quickest way to get the familiar identifiers all recognizable by their usual short names. This is the method that we have been using in this class. The problem is that it introduces a lot of other less familiar names as well, increasing the chances of an accidental name clash with one of your variables.

For these reasons instructors or bosses in future classes or programming jobs may requiring rules such as the following.

RULES FOR USING namespace

1) No using namespace outside of a block (no global definition of it)
Any "using" must be in a block.

2) Use the most restricting from that is convenient. This is known as using the full name.

```
{
   cout << x;                    use    std::cout << x;
}

{
  Multiple use of cout                  using std::cout;
 and endl                               using std::endl;
        }
```

Exceptions

Exceptions are signals that errors or unexpected events occurred while the program is running. We have tested for some errors thus far by using the conditional statement. For example the following (which determines the average grade of a class) could be used for a dividing by 0 error:

```
average=divide(totalGrades, numOfGrades);


float divide (int numerator, int denominator)
if ( denominator == 0)
   cout << "Error:  Cannot divide by zero.\n";
else
        return float(numerator)/denominator;
```

Although this is fine for simple programs, complex programs require a more sophisticated method for dealing with errors or unusual events.

Exceptions are very good ways of handling complex error conditions. An exception is a value or an object that signals an error. When an error occurs, an exception is "**thrown.**" This simply means some specific code (called an exception handling routine) is executed to handle that particular error. The divide by 0 error above could be handled as follows:

```
float divide (  int  numerator, int denominator)
{
        if (denominator==0)
                throw "Error:  Cannot divide by zero.\n";  // This is known as the throw point
        else
                return float (numerator) /denominator;
}
```

The throw keyword is followed by an argument which in this case was a string that contained an error message.
To handle an exception, a program will have a **try/catch** construct. A `try` block contains any instructions that might cause an exception either directly or indirectly. It starts with the keyword `try` The try block is followed by the `catch` block which begins with the keyword `catch` followed by the declaration of an exception parameter which enclosed in parenthesis.

The try/catch block for the dividing by 0 exception is as follows:

```
try
{
        average = divide(totalGrades,numOfGrades)
```

```
        cout << “The average grade is “ << average << endl;
}

catch   (char *exceptionString)
{
        cout << exceptionString;
}
```

When the function divide is called within the try block, if the second parameter is 0 then a throw is activated which is caught by the catch block. The throw gives an argument, which, as can be seen from the catch parameter list, is a string.

```
float divide (  int  numerator, int denominator)
{
        if (denominator==0)
                throw “Error:  Cannot divide by zero.\n”;  // This is known as the throw point
        else
                return float (numerator) /denominator;
}
```

Example program 13.1 gives a complete program using these concepts

```
#include <iostream>

// Function prototype

float divide (int, int);

int main()
{
      int value1, value2;
      float quotient;
      std::cout << "Enter the first of two numbers: ";
      std::cin >> value1 ;
      std::cout <<"Enter the second of two numbers: ";
      std::cin >> value2;

      try
      {
            quotient = divide(value1,value2);
            std::cout << "The quotient is " << quotient << std::endl;
      }

      catch (char *exceptionString)
      {
            std::cout<<exceptionString;
      }

      std::cout << "end of the program.\n";
}

      float divide (int numerator, int denominator)
      {
```

```
            if (denominator ==0)
                  throw "ERROR:  Cannot divide by zero.\n";
            else
                  return float(numerator)/denominator;
      }
```

Notice that we used std:: before each cin, cout and endl. In this case we used the full name option that included the scope resolution operator each time we used those values.

Multiple Exceptions

Exception handling routines are useful in many applications. Suppose we want to check grades that range from 0 to 100. We can use multiple exceptions where we test for several different types of errors and signal which one has occurred. In this case we can create one type of throw if the user enters a negative number and another type of throw if the user enters a number greater than 100.

In this next example we use Object Oriented programming techniques (with a header file) learned from the previous lessons.

We create a header file that defines a class called `IntRange` that has three integer private data members: `input, lower, upper`. The variable `input` serves as the holder of the number the user will input as a grade. `lower` and `upper` serve as the boundaries of the grades. In has two empty class declarations: one for `TooLow` and one for `TooHigh`. The member functions include the constructor that gives values to `lower` and `upper` and a `getInput` function that allows the user to input a value which is then checked. If the value is too low than it throws to the `TooLow` class, if the value is too high than it throws to the `TooHigh` class.

Example 13.2

```
#ifndef INTRANGE_H
#define INTRANGE_H

class IntRange
{
private:
      int input;  // For user input
      int lower;  // Lower limit of range
      int upper;  // upper limit of range

public:
      // Exception classes

      class TooLow
      {
      };
      class TooHigh
      {
      };
// member functions

      IntRange(int low, int high)  // Constructor
```

```
	{
		lower =low;
		upper=high;
	}

	int getInput()
	{
		std::cin >> input;
		if (input < lower )
			throw TooLow();
		else if (input > upper)
			throw TooHigh();
		return input;
	}
};
#endif
```

The client program contains two catches: one to handle a value too low and another to handle a value too high. In this case we used using `std::cout; using std::cin;` and using `std::endl;` inside the `main` so we did not have to use the full name in the main program for each `cin, cout` and `endl`.

Example program 13.3

```
#include <iostream>
#include "IntRange.h"



int main()
{
	using std::cout;
	using std::cin;
	using std::endl;

	IntRange range(0,100);
	int value;
	cout << "Enter a grade in the range of 0 to 100 ";
	try
	{
		value=range.getInput();
		cout << "You entered " << value << endl;
	}


	catch (IntRange::TooLow)
	{
		cout<<"That value is too low\n";
	}
	catch (IntRange::TooHigh)
	{
		cout << "That value is too high"<<endl;
	}

	cout << "End of the program\n";
}
```

Extracted Information

At times we may want an exception to pass back information to the exception handler. In other words we may want the grade that was out of range to be printed with the statement that indicated that the grade was out of range. We can alter the IntRange class to handle that. The following header file does just that. Note that it now does not indicate too low or too high but rather prints the value that was input.

Example program 13.4

```
#ifndef INTRANGE_H
#define INTRANGE_H

class IntRange
{
private:
      int input;  // For user input
      int lower;  // Lower limit of range
      int upper;  // upper limit of range

public:
      // Exception class

      class OutOfRange
      {
      public:
            int value;
            OutOfRange(int i)
            {
                  value = i;
            }

      };
// member functions

      IntRange(int low, int high)  // Constructor
      {
            lower =low;
            upper=high;
      }

      int getInput()
      {
            std::cin >> input;
            if (input < lower )
                  throw OutOfRange(input);

            return input;
      }
};
```

```
#endif
```

The client file that uses this is listed here.
Example program 13.5

```
#include <iostream>
#include "IntRange.h"

int main()
{
	using std::cout;
	using std::cin;
	using std::endl;

	IntRange range(0,100);
	int value;
	cout << "Enter a grade in the range of 0 to 100 ";
	try
	{
		value=range.getInput();
		cout << "You entered " << value << endl;
	}

	catch (IntRange::OutOfRange ex)
	{
		cout<<"That value " << ex.value << " is out of range.\n";
	}

	cout << "End of the program\n";
}
```

If a try/catch construct contains no catch blocks with the correct data type parameter or if the exception is thrown from outside a try block, the program will abort. This is why it is important for exceptions and their handling routines to be incorporated into code. The exception can be handled and the rest of the program can proceed.

Enumerated data types

An enumeration data type is a type that is defined by the user in which the constant identifiers (called enumerators) are explicitly listed. Since they are defined by the user, they are often called user-defined data types. Since these data types are often used by all functions in a program they are usually declared in the global section.

Example:

```
enum BirdType {BLUEJAY, CARDINAL, ROBIN, SEAGULL, SWALLOW};
enum LetterGradeType {A, B, C,D,F};
```

To create an enumerated type begin with the reserved word `enum` followed by a user name for the new data type followed by a set of braches that includes all the possible data types in some sequence which defines its order in the set. It is custom to label data types with an upper case letter for the first letter of each word and lower case for all others.

MAIN

```
BirdType birdie;
LetterGradeType grade;
```

NOTICE: `BirdType` is an enumerated DATA TYPE. It is a type of data just like `integer` or `float`. It is a data type declared in the program. It can be used to define variables (such as `birdie` in the above example).

`birdie` can contain any of the enumerators (constants) listed in data type `BirdType`. `grade` can contain any of the enumerators listed in data type `LetterGradeType`.

The enumerators of an enumeration type can be used in a program just like any other constant.

```
birdie = ROBIN;
grade = A;
if (grade == B)
```

NOTE: `grade` does NOT contain the character `A`. Grade contains a value of the data type `LetterGradeType`, the enumerator `A`. Thus there are not single quotes.

Enumerators are ordered by the way they are listed. In a relational expression, enumerators are evaluated exactly as characters would be: by the ordering of the enumerators.

There is a problem! We cannot print out the values of an enumeration type directly. We must convert the value from the enumeration type into a string that corresponds to the appropriate enumerator.

Example:

```
switch (birdie)
{
      case BLUEJAY: cout << “BlueJay”;
                      break;
      case CARDINAL: cout << “Cardinal”;
                    break;
}
```

Value of an enumeration type cannot be read; they must be set in the program. The name of the enumeration type can be used to convert, or type cast a number representing the position of an enumerator in the listing of the data type into the enumerator in that position.

Ex. `birdie = BirdType(0)` stores `BLUEJAY` into variable `birdie` because `BLUEJAY` is in the 0^{th} position of the enumerators. You can use this technique to input values of enumeration types. The user can be given a menu showing the enumerators and asked to key in the number representing the enumerator they wish to input. The number is read and type cast into the enumerator.
Example:

```
        char animal;
        cout << “Input a B for Bluejay”<<endl;
                << “C for Cardinal\n”
                << “R for Robin\n”
                << “S for Seagull\n”
                << “W for Swallow\n”;
        cin >> animal;

switch(animal)
        {
                case ‘B’: birdie = BLUEJAY;
                        break;
                case ‘C’: birdie = CARDINAL;
                        break;
                case ‘R’: birdie = ROBIN;
                        break;
                case ‘S’: birdie = SEAGULL;
                        break;
                case ‘W’: birdie = SWALLOW;
                        break;
        }
```

Lab 13 Namespaces, Event Handling Routines & Enumerated Data

The purpose of the lab is to give the student experience in developing and working with namespaces and exception handling routines.

Fill in the Blank Questions

1. Namespaces deal with the concept of _____.
2. All the identifiers in the ANSI standard header file `iostream` are part of the _____namespace.
3. They symbol ____ before an instruction indicates that this is a compiler directive.
4. _____Exceptions are signals that errors or unexpected events occurred.
5. Exceptions are handled in a _____construct.
6. A(n) _____ routine is executed to handle an exception.
7. When an error occurs, an exception is _____which means that some specific code is executed to handle that particular error or event.
8. A(n) _____data type is defined by the user in which the constant
9. Constant identifiers called _____ are explicitly listed.
10. We cannot print out the values of an enumeration type directly. We must first convert the value from the enumeration type into a _____ that corresponds to the appropriate enumerator.

NOTE: In all of these labs you will not be allowed to use the global statement:
`using namespace std;`

Problem 13.1 Enumeration data types

Look at the following program:

```
#include <iostream>

enum BirdType {BLUEJAY, CARDINAL,ROBIN,SEAGULL,SWALLOW};


int main()
{
        BirdType birdie;    // birdie has an enumerated data type that can
```

```
                        // take on any value listed in the set:  BLUEJAY
                        // or CARDINAL or ROBIN or SEAGULL or SWALLOW
    char birdLetter;    // birdLetter is used to input user's choice

    std::cout<< "Please select your favorite bird from the following list:\n"
             <<  " B for Bluejay\n"
             <<  " C for Cardinal\n"
             <<  " R for Robin\n"
             <<  " S for Seagull\n"
            <<  " W for Swallow\n";


    std::cin>> birdLetter;


    switch(birdLetter)
    {
    case 'B' :
    case 'b':                   birdie=BLUEJAY;
                                break;
    case 'C':
    case 'c':                   birdie=CARDINAL;
                                break;
    case 'D' :
    case 'd':                   birdie=ROBIN;
                                break;
    case 'S':
    case 's':                   birdie=SEAGULL;
                                break;
    case 'W':
    case 'w':                   birdie=SWALLOW;
                                break;
    }

    if (birdie == BLUEJAY)
            std::cout<<"Your favorite bird is the Bluejay" << std::endl;
    else if (birdie==CARDINAL)
            std::cout<<"Your favorite bird is the Cardinal" << std::endl;
    else if (birdie==ROBIN)
            std::cout<<"Your favorite bird is the Robin" << std::endl;
    else if (birdie==SEAGULL)
            std::cout<<"Your favorite bird is the Seagull" << std::endl;
    else if (birdie==SWALLOW)
            std::cout<<"Your favorite bird is the Swallow" << std::endl;
    return 0;
}
```

Exercise 1: Create a project and write the above program into a file called bird.cpp. Then run the program using the menu items. Run it again using a letter not given in the menu. What happens?

Exercise 2: Alter the program so that it has try / catch blocks (an exception handling routine) that will indicate if the user did not enter one of the appropriate menu choices.

Sample Run:

Please select your favorite bird from the following list:
B for Bluejay
C for Cardinal
R for Robin
S for Seagull
W for Swallow
z
Error: Not a proper selection for the menu

Problem 13.2 Exception handling routines Multiple Exceptions

Bring in `waterTemperatur.cpp` from the Lab 13 folder. The code is as follows:

```
#include<iostream>
// This program asks the user to input the temperature of water
// If the temperature of the water is 65 or more than the water will
// be suitable for swimming, otherwise it will not.

int main()
{
      double waterTemp;  // temperature of water input by user

      cout << "Please input the temperature of the water"<< endl;
      cin >> waterTemp;

      if (waterTemp >= 65)
            cout <<"The water temperature is fine for swimming" << endl;
      else
            cout <<"The water temperature is not good for swimming" << endl;
}
```

Exercise 1: Just by looking at the program will this code run properly? If not please fix it so that it will run properly and produce the following sample output:
Sample run 1:

Please input the temperature of the water
78
The water temperature is fine for swimming

Sample run 2:

Please input the temperature of the water
55
The water temperature is not good for swimming

Exercise 2: The program as it now stands will allow the user to input and temperature. We know that water becomes ice at 32 degress and steam at 212 degrees. Change the program to include two exceptions: one to handle the case where temperature was equal to or over 212 in which case the exception will indicate that steam not water was entered and the other case where temperature was equal or less than 32 in which case the exception will indicate that ice not water was entered.

Sample run 1:
Please input the temperature of the water
245
Error: You entered steam not water!

Sample run 2:
Please input the temperature of the water
23
Error: You entered ice not water!

Sample run 3:

Please input the temperature of the water
55
The water temperature is not good for swimming

Student Generated Code Assignments

Student Generated Code Program 13.3

Write a program that has an enumerated data type consisting of the letter grades A B C D F. The user will input an integer to select one of those grades: integer 1 for A 2 for B 3 for C 4 for D and 5 for F. The program will continue to ask if the user if another grade is to be entered until the user responds with something other than a y. Note: We want the program to continue if the person inputs an incorrect selection and not to just terminate.
The output will just indicate the letter grade.

Sample Run:

Please select a grade from the following:
1 for A
2 for B
3 for C

4 for D
5 for F
3
You got a C--2.0
Would you like another grade to enter: enter y for yes and n for n
y
Please select a grade from the following:
1 for A
2 for B
3 for C
4 for D
5 for F
6

That was an incorrect choice.
Please select a grade from the following:
1 for A
2 for B
3 for C
4 for D
5 for F
1
You got a A--4.0
Would you like another grade to enter: enter y for yes and n for n
N
Thank-you Good-bye.

Student Generated Code Program 13.4 Classes, header files, operator overloads and exceptions

Write a program that will keep track of package characteristics for a shipping company. It should declare a `Shipping` class that includes 5 integer member variables: `packageNum, length, width, height`, and `weight`. Dimensions are in inches and weight is in ounces. Include a default constructor that sets `packageNum` to -99 and other member variables to 1. Include a regular constructor to initialize all the values with user-supplied data. The class should have functions to `set` and `get` (i.e., place and retrieve) each of the data members. Include a function to calculate package volume and include overloaded operators ==, >, and < to compare two packages. Two packages are "equal" if they have the same volume and the same weight. A package is "greater than" another if it has a greater volume or has the same volume but a greater weight. Have the program create a pair of `Shipping` objects, and have their data members (all five for each one) input from the user. The program should output their characteristics, including the volume of each and then indicate if they are equal or which is greater. The program should also include a try/catch block that has multiple throws. A throw for each user input that has a negative value. For example if the user inputs a negative number for a packageNum the exception should read: Error: Can't have a negative package number. It should have a different error message for each member data that is input as a negative number.

Sample run 1

Please input the first package number
101
Please input the first package length
12
Please input the first package width
8
Please input the first package height
4
Please input the first package weight
7
Please input the second package number
102
Please input the second package length
8
Please input the second package width
8
Please input the second package height
6
Please input the second package weight
7
package number: 101 dimensions: 12 x 8 x 4 weight: 7 oz.
package number: 102 dimensions: 8 x 8 x 6 weight: 7 oz.
The volume of package 1 is 384
The volume of package 2 is 384
The packages are equal

Sample run 2

Please input the first package number
101
Please input the first package length
-4
Error: Can't have a negative length

Lesson 13 Summary

Namespaces deal with the concept of scope. You recall that scope defines areas in the program where identifiers (variables, constants etc.) are active. Namespaces allow the programmer to create a scope for global variables. This is necessary when many programmers or multiple software packages are involved. More than one programmer, or software package, may use the same name for a variable thus creating a conflict of names. This problem can be solved if each programmer, or software vendor, places each set of its classes, and identifiers in a separate place called a namespace that has its own unique name.

```
namespace namespace_name
{
        Declarations
}
```

RULES FOR USING namespace

No using namespace outside of a block (no global definition of it)
Any "using" must be in a block.

Use the most restricting from that is convenient. This is known as using the full name.

```
{
   cout << x;                          use     std::cout << x;
}



{
  Multiple use of cout                         using std::cout;
 and endl                                      using std::endl;
        }
```

Exceptions

Exceptions are signals that errors or unexpected events occurred while the program is running.

Exceptions basically have three parts: a try section a catch section and one or more throw statements inside a try block that when activated will call the catch section.

Example:

```
try
{
        average = divide(totalGrades,numOfGrades)
        cout << "The average grade is " << average << endl;
}

catch   (char *exceptionString)
{
```

```
        cout << exceptionString;
}
```

When the function divide is called within the try block, if the second parameter is 0 then a throw is activated which is caught by the catch block. The throw gives an argument, which, as can be seen from the catch parameter list, is a string.

```
float divide (  int  numerator, int denominator)
{
        if (denominator==0)
                throw "Error:  Cannot divide by zero.\n";  // This is known as the throw point
        else
                return float (numerator) /denominator;
}
```

Lesson 14: Advanced File Operations

Review of Text Files

A **file** is a collection of information stored (usually) on a disk. Files, just like variables, have to be defined in the program. The data type of a file depends on whether it is used as an input file, output file, or both. Output files have a data type called `ofsteam`, input files have a data type of `ifstream`, and files used as both have a data type of `fstream`. We must add the `#include<fstream>` directive whenusing files.

Examples:

```
ofstream outfile;       // defing outfile as an output file
ifstream infile;              // defining infile as an input file
fstream datqafile;            // defining datafile to be both an input and output
file.
```

After their definition, files must still be opened, used (information stored to or data read from the file), and then closed.

Opening Files

A file is opened with the open function. This ties the logical name of the file that is used in the definition to the physical name of the file used in the secondary storage device (disk). The statement `infile.open("payroll.dat");` opens the file `payroll.dat` and lets the program know that `infile` is the name by which this file will be referenced within the program. If the file is not located in the same directory as the C++ program, the full path(drive, etc.) MUST be indicated: `infile.open()"c:\\payroll.ldat");` This tying of the logical name `infile` with the physical name payroll.dat means that wherever `infile` is used in the program, data will be read from the physical file `payroll.dat`. A program should check to make sure that the physical file exists. This can be done by a conditional statement.

Example:

```
ifstream infile;
infile.open("payroll.dat");
if (!infile)
{
        std::cout << "Error opening file.  It may not exist where  indicated.\n";
        return 1;
}
```

Note: In the above example, return 1 is used as an indicator of an abnormal occurrence. In this case the file in question could not be found.

Reading from a File

Files have an "invisible" end of line marker at the end of each line of the file. Files also have an

“invisible” end of file marker at the end of the file. When reading from an input file within a loop, the program must be able to detect that marker as the sentinel data (data that meets the condition to end the loop). There are several ways to do this.

Example Program 14.1

```
#include <fstream>
#include <iostream>
#include <iomanip>

int main()
{

	std::ifstream infile;		// defining an input file
	std::ofstream outfile;		// defining an output file

	infile.open("payroll.dat");
	// This statement opens infile as an input file.
	// Whenever infile is used, data from the file payroll.dat will be read

	outfile.open("payment.out");
	// This statement opens outfile as an output file.
	// Whenever outfile is used, information will be sent to the file payment.out

	int hours;				// The number of hours worked
	float payRate;				// The rate per hour paid
	float net;				// The net pay

	if(!infile)
	{
	std::cout << "Error opening file.\n";
	std::cout << "Perhaps the file is not where indicated.\n";
	return 1;
	}

	outfile << std::fixed << std::setprecision(2);
	outfile << "Hours	Pay Rate	Net Pay" << std::endl;

	infile >> hours;		// priming the read

	while(infile)
	{
		infile >> payRate;
		net = hours * payRate;
		outfile << hours << std::setw(10) << "$ " << std::setw(6)
			<< payRate << std::setw(5)
			<< "$ " << std::setw(7) << net << std::endl;
		infile >> hours;
	}
	infile.close();
	outfile.close();
	return 0;
}
```

Notice the statement `outfile << std::fixed << std::setprecision(2);` This shows that the format procuedures learned for cout can be used for output files as will. Remember that `setw(x)` can be used as long as the `iomanip` header file is included.
This program assumes that a data file exists that contains an undetermined number of records with each record consisting of two data values, `hours` and `payRate`. Suppose that the input data file (`payroll.dat`) contains the following:
40 10.00
30 6.70
50 20.00

The program will produce the following output file (`payment.out`).

Hours	**Pay Rate**	**Net Pay**
40	**$ 10.00**	**$ 400.00**
30	**$ 6.70**	**$ 201.00**
50	**$ 20.00**	**$1000.00**

The input file contains data for one employee on each line. Each time through the `while` loop, information is processed for one employee. The loop executes the same number of times as there are lines (employee records in this case) in the data file. Since there are two items of data for each line (`hours` and `payRate`), these items are read in each time through the loop. Notice that one of the input variables was input before the `while` loop. This is called "priming the read." Input can be thought of as a stream of values taken one at a time. Before the while loop condition can be tested, there has to be something in that "stream." We **prime** the read by reading in at least one variable before the loop. Observe that the statement `infile >> hours;` is given twice in the program: once before the input loop and as the last statement in the loop. The other item, `payRate`, is read in at the very beginning of the loop. This way each variable is read each time through the loop. Also notice that the heading of the output file is printed outside the loop before it is executed.

There are other ways of determining that the end of the file has been reached. The `eof()` function reports when the end of a file has been encountered. The loop in Example Program 14.1 can be replaced with the following:

```
while(!infile.eof())
{
infile >> payRate;
net = hours * payRate;
outfile << hours << std::setw(10) << "$ " << std::setw(6) << payRate <<
std::setw(5)
<< "$ " << std::setw(7) << net << std::endl;
infile >> hours;
}
```

In addition to checking to see if a file exists, we can also check to see if it has any data in it. The following code checks first if the file exists and then if it is empty.

```
infile.open("sample2.dat");
```

```
if(!infile)
std::cout << "file does not exist" << std::endl;
else if(ch = infile.peek()) == EOF)
std::cout << "File is empty" << std::endl;
else
```
//rest of program

The `peek` function actually looks ahead in the file for the next data item, in this case to determine if it is the end of file marker. `ch` must be declared as char data type.

Since the `peek` function looks "ahead" for the next data item, it could be used to test for end of file in reading values from a file within a loop without priming the read.

The following program accomplishes the same thing as Example Program 14.1 without priming the read. The portions in bold differ from Example Program 14.1

Example Program 14.2

```
#include <fstream>
#include <iostream>
#include <iomanip>

int main()
{

std::ifstream infile;         // defining an input file
std::ofstream outfile;        // defining an output file

infile.open("payroll.dat");
// This statement opens infile as an input file.
// Whenever infile is used, data from the file payroll.dat will be read

outfile.open("payment.out");
// This statement opens outfile as an output file.
// Whenever outfile is used, information will be sent to the file payment.out

int hours;                          // The number of hours worked
float payRate;                      // The rate per hour paid
float net;                          // The net pay
char ch;                            // ch is used to hold the nex value
                                    // (read as character) from the file
if(!infile)
{
std::cout << "Error opening file.\n";
std::cout << "Perhaps the file is not where indicated.\n";
return 1;
}

outfile << std::fixed << std::setprecision(2);
outfile << "Hours      Pay Rate       Net Pay" << std::endl;

infile >> hours;        // priming the read
```

while((ch=infile.peek()) != EOF)

```
{
infile >> payRate;
net = hours * payRate;
outfile << hours << std::setw(10) << "$ " << std::setw(6) << payRate <<
std::setw(5)
<< "$ " << std::setw(7) << net << std::endl;
infile >> hours;
}
infile.close();
outfile.close();
return 0;
}
```

Output Files

Output files are opened the saem way: `outfile.open("payment.out")`. Whenever the program writes to outfile, the information will be placed in the physical file `payment.out`. Notice that the program generates a file stored in the same location as the source file. The user can indicate a different location for the file to be stored by indicating the full path(drive, etc.).

Files Used for Both Input and Output

A file can be used for both input and output. The `fstream` data type, which is used for files that can handle both input and output, must have a file access flag as an argument to the `open` function so that the mode, input or output, can be determined. There are several access flags that are used to indicate the use of the file. The following chart lists frequently used access flags.

Flag mode	Meaning
`ios::in`	Input mode. The file is used for "reading" information. If the file does not exist, it will not be created.
`ios::out`	Output mode. Information is written to the file. If the file already exists, its contents will be deleted.
`ios::app`	Append mode. If the file exists, its contents are preserved and all output is written to the end of the file. If it does not exist, then the file will be created. Notice how this differs from `ios::out`.
`ios::binary`	Binary mode. Information is written to or read from in pure binary format (discussed later in this lesson).

Sample:

```
#include <fstream>
```

```
int main()
{

      std::fstream test ("grade.dat", std::ios::out);
      // test is defined as an fstream file first used for output
      // ios:: out is the file access flag

      // code of program goes here
      // the code will put values in the test file

      test.close();                  // close the file as an output file
      test.open("grade.dat", std::ios::in);
      // the same file is reopened now as an input file
      // ios::in is the file access flag

      // other code goes here
      test.close();           // close the file
return 0;
}
```

In the following example, we check for a file's existence before opening it. We first attempt to open the file for input. If the file does not exist, then the open operation fails and we open it as an output file.

Example Program 14.3

```
#include <fstream>
#include <iostream>


int main()
{

      std::fstream dataFile;
      dataFile.open("grades.txt", std::ios::in);
      if (dataFile.fail())
            // The fail directive indicates the file did not open
            // since it does not exist
      {     dataFile.open("grades.txt", std::ios::out);
            // file is processed here:  data sent to the file
      }
      else  // the file already exists.
      {
            std::cout << "The file grades.txt already exist. \n";
                  // process file here
            dataFile.close();
      }

return 0;
}
```

Just as `cin >>` is used to read from the keyboard and `cout <<` is used to write to the screen, `filename >>` is used to read from the file filename and `filename <<` is used to write to the file filename.

Closing a File

Files should be closed before the program ends to avoid corrupting the file and/or losing valuable data.

```
infile.close();
outfile.clsoe();
dataFile.close();
```

Passing Files as Parameters to Functions

Files can be passed as parameters just like variables. Files are always passed by reference. The & symbol must be included after the data type in the function heading and prototype.

Example:

```
void GetData(ifstream& infile, ofstream& outfile);     // prototype of function
                                                       // with files as parameters
void GetData(ifstream& infile, ofstream& outfile)      // heading of function with
                                                       // files as parameters
```

Review of Character Input

Recall that each file has an end of line marker for each line as well as the end of file marker at the end of the file. Whenever whitespace (blanks, newlines, controls, etc.) is part of a file, a problem exists with the traditional >> operator in inputting character data. When reading input characters into a string object, the >> operator skips any leading whitespace. It then reads successive characters into the character array, stopping at the first trailing whitespace character (which is NOT "consumed," but rather which remains as the next character to be read in the file). The >> also takes care of adding the null character to the end of the string. Stopping at the first trailing whitespace can create a problem for names containing spaces. A program reading first names into some string variable (array of characters) has a problem reading a name like Mary Lou since it has a blank space in it. The blank space between Mary and Lou causes the input to stop when using the >> operator. The get function can be used to input such strings.

```
infile.get(firstname, 20);
```

The get function does NOT skip leading whitespace characters but rather continues to read characters until it either has read, in the example above, 19 characters or it reaches the newline character \n (which it does NOT consume). The last space is reserved for the null character.

Since the get function does not consume the end of line character, there must be something done to consume it so that a new line can be read.

Example: Given the following data file

```
Mary Lou <eol>
Becky <eol>
```

```
Debbie <eol>
<eof>
```

Note: Both the <eol> and <eof> are NOT visible to the programmer or user. There are several options for reading and printing this data.

```
char dummy;                    // created to read the end of line character
char firstname[80];           // array of characters for the first name
outfile << "Name " << std::endl;
infile.get(firstname,80);     // priming the read    inputs the first name

while(infile)
{
infile.get(dummy);            // reads the end of line character into dummy
outfile  <<  firstname  <<  endl;    //outputs the name
infile.get(firstname,80);            //reads the next name
}
```

In the above example, dummy is used to consume the end of line character.
`input.get(firstname,80);` reads the string Mary Lou and stops just before reading the <eol> end of line character. The `infile.get(dummy)` gets the end of line character into dummy.

Another way to do this is with the `ignore` function, which reads characters until it encounters the specific character it has been instructed to look for or until it has skipped the allotted number of characters, whichever comes first. The statement `infile.ignore(81,'\n')` skips up to 81 characters stopping if the new line `'\n'` character is encountered. This newline character IS consumed by the function, and thus there is no need for a dummy character variable.

Example:

```
char firstname[80];

outfile << "Name " << std::endl;
infile.get(firstname,80);

while(infile)
{
infile.ignore(81,'\n'); // read and consume the end of line character
outfile << firstname << std::endl;
infile.get(firstname,80);
}
```

Still another way to read in lines containing whitespace characters is with the getline member function. It needs neither a dummy variable nor the ignore function.
Example:
char firstname[80];
outfile << "Name " << endl;
infile.getline(firstname,81); //We use a number 1 greater than the total
//amount to read. Lines contain 80 characters

```
while(infile)
{
outfile << firstname << endl;
infile.getline(firstname,81);
}
```

The following example program shows how names with embedded whitespace alone with numeric data can be processed. Parts in bold indicate the changes from Example Program 12.2. Assume that the `payroll.dat` file has the following information:

```
John Brown      40      10.00
Kelly Barr      30       6.70
Tom Seller      50      20.00
```

The program will produce the following information in `payment.out`:

```
Name            Hours      Pay Rate       Net Pay
John Brown        40        $  10.00   $  400.00
Kelly Barr        30        $   6.70   $  201.00
Tom Seller        50        $  20.00   $ 1000.00
```

Example Program 14.4

```
#include <fstream>
#include <iostream>
#include <iomanip>

const int MAX_NAME = 11;

int main()
{

std::ifstream infile;         // defining an input file
std::ofstream outfile;        // defining an output file

infile.open("payroll.dat");
// This statement opens infile as an input file.
// Whenever infile is used, data from the file payroll.dat will be read

outfile.open("payment.out");
// This statement opens outfile as an output file.
// Whenever outfile is used, information will be sent to the file payment.out

int hours;                          // The number of hours worked
float payRate;                      // The rate per hour paid
float net;                          // The net pay
char ch;                            // ch is used to hold the nex value
                                        // (read as character) from the file
char name[MAX_NAME];                // array of characters for the name of
```

```
                                        // a student, with at most 10 characters
if(!infile)
{
std::cout << "Error opening file.\n";
std::cout << "Perhaps the file is not where indicated.\n";
return 1;
}

outfile << std::fixed << std::setprecision(2);
outfile << "Name            Hours      Pay Rate       Net Pay" << std::endl;



while((ch=infile.peek()) != EOF)
{
infile.get(name,MAX_NAME);              // gets names with blanks
infile >> hours;
infile >> payRate;
infile.ignore(81,'\n');
net = hours * payRate;
outfile << name << std::setw(10) <<hours << std::setw(10) << "$ " << std::setw(6) <<
payRate << std::setw(5)
<< "$ " << std::setw(7) << net << std::endl;

}
infile.close();
outfile.close();
return 0;
}
```

Another way to read in lines containing whitespace characters is with the `getline` member function.

Example:

```
char firstName[80];
outfile << "Name " << std::endl;
infile.getline(firstName,81);

while (infile)
{
        outfile << firstName << std::endl;
        infile.getline(firstName,81);
}
```

Binary Files

So far all the files we have talked about have been text files, files formatted as ASCII text. Even numbers written to a file with the << operator are changed to ASCII text. ASCII is a code that stores every datum (letter of the alphabet, digit, punctuation mark, etc.) as a character with a unique number.

Although ASCII text is the default method for storing information in files, we can specify that we want to store data in pure binary format by "opening" a file in binary mode with the `ios::binary` flag. The `write` member function is then used to write binary data to the file. This method is particularly useful for transferring an entire array of data to a file.

Example:

fstream test("grade.dat", ios::out | ios::binary); // This declares and opens
// the file test as an output binary file

int grade[arraysize] = {98, 88, 78, 77, 67, 66, 56, 78,98, 56}; // creates and initializes an integer
// array

test.write((char*)grade, sizeof(grade)); // write all values of array to file

test.close(); // close the file

`test.write((char*)grade, sizeof(grade));` in the above example calls the `write` function. The name of the file to be written to is `test`. The first argument is a character pointer pointing to the starting address of memory, in this case to the beginning of the `grade` array. The second argument is the size in bytes of the item written to the file. `sizeof` is a function that determines that size.

The following sample program initializes an array and then places those values into a file as binary numbers. The program then adds 10 to each element of the array and prints those values to the screen. Finally the program reads the values from the same file and prints them. These values are the original numbers. Study the program and its comments very carefully.

Example Program 14.5

```
#include <fstream> //directive needed to use files
#include <iostream>


const int ARRAYSIZE = 10;
int main()
{

std::fstream test("grade.dat", std::ios::out| std::ios::binary);
 // note the use of | to separate file access flags
int grade[ARRAYSIZE] = {98, 88, 78, 77, 67, 66, 56, 78, 98, 56};
int count;  // loop counter

test.write((char*)grade, sizeof(grade));  // write values of array to file
test.close(); //close file

//  now add 10 to each grade

std::cout << "The values of grades with 10 points added" << std::endl;
```

```
for (count = 0; count < ARRAYSIZE; count++)
{
grade[count] = grade[count] + 10;   //  This adds 10 to each element of array
std::cout << grade[count] << std::endl;   // write the new values to the screen
}
test.open("grade.dat", std::ios::in); //  reopen the file but now as an input file

test.read((char*) grade, sizeof(grade));

/*    The above statement reads from the file test and places the values found
into the grade array. As with the write function, the first argument is a
character pointer even though the array itself is an integer. It points to the
starting address in memory where the file information is to be transferred.
*/
std::cout << "The grades as they were read into the file" << std::endl;

for (count = 0; count < ARRAYSIZE; count++)
std::cout << grade[count] << std::endl;   //  writes original values to the screen
test.close();
return 0;
}
```

The output to the screen from this program is as follows:

The values of grades with 10 points added
108
98
88
87
77
76
66
88
108
66
The grades as they were read into the file
98
88
78
77
67
66
56
78
98
56

Files and Records

Files are often used to store records. A “field” is one piece of information and a “record” is a group of

fields that logically belong together in a unit.

Example:	**Name**	**Test1**	**Test2**	**Final**
	Brown	89	97	88
	Smith	99	89	97

Each record has four fields: `Name, Test1, Test2` and `Final`. When records are stored in memory, rather than in files, C++ structures provide a good way to organize and store them.

```
struct Grades
{
        char name[10];
        int test1;
        int test2;
        int final;
};
```

An identifier defined to be a `Grades` structure could hold one record.

The `write` function, mentioned in the previous section, can be used to write records to a file.

```
fstream test("score.dat", ios::out|ios::binary);
Grades student;                 // defines a structure variable

// code that gets information into the student record
test.write((char *) & student, sizeof(student));
```

The test.write function used to write a record stored as a struct is similar to the write function used for an array with one big difference. Notice the inclusion of (&). Why is this necessary here and not with writing an array? This is pass by reference which requires the (&) symbol. Arrays are passed by pointer. The following example takes records from the user and stores them into a file.

Example Program 14.6

```
#include <fstream >
#include <iostream >
#include <cctype>               // for toupper function

const int NAMESIZE = 31;
struct Grades                  //  declaring a structure
{
char name[NAMESIZE];
int test1;
int test2;
int final;
};

int main ()
```

```
{
using namespace std;
fstream tests("score.dat", ios::out| ios::binary);
//  defines tests as an output binary file.

Grades student;     // defines student as a record (struct)
char more;        //  used to determine if there is more input
do
{
cout << "Enter the following information" << endl;
cout << "Student's name: ";
cin.getline(student.name,NAMESIZE);
cout << "First test score :";
cin >> student.test1;
cin.ignore(); //ignore rest of line

cout << "Second test score: ";
cin >> student.test2;
cin.ignore();

cout << "Final test score: ";
cin >> student.final;
cin.ignore();

// write this record to the file
tests.write((char *)&student, sizeof(student));

cout << "Enter a y if you would like to input more data" << endl;
cin >> more;
cin.ignore();
} while (more == 'y');
tests.close();
return 0;
}
```

Random Access Files

All the files studied thus far have performed **sequential file access**, which means that all data is read or written in a sequential order. If the file is opened for input, data is read starting at the first byte and continues sequentially through the file's contents. If the file is opened for output, bytes of data are written sequentially. The writing usually begins at the beginning of the file unless the `ios::app` mode is used, in which case data is written at the end of the file. C++ allows a program to perform **random file access**, which means that any piece of data can be accessed at any time. The old technology of a cassette tape is an example of a sequential access medium. To listen to the songs on a tape, one had to listen to them in the order they were recorded or fast forward (reverse) through the tape to get to a particular song. A CD has properties of a random access medium. One simply jumps to the track where a song is located. It is not truly random access, however, since one cannot jump to the middle of a song.

There are two file stream member functions that are used to move the read/write position to any byte in the file. The `seekp` function is used for output files and `seekg` is used for input files.

Example:

```
dataOut.seekp(30L, ios::beg);
```

This instruction moves the marker position of the file called `dataOut` to 30 positions from the beginning of the file. The first argument, `30L` (L indicates a long integer), represents an offset (distance) from some point in the file that will be used to move the read/write position. That point in the file is indicated by the second argument (`ios::beg`). This access flag indicates that the offset is calculated from the beginning of the file. The offset can be calculated from the end (`ios::end`) or from the current (`ios::cur`) position in the file.

If the `eof` marker has been set (which means that the position has reached the end of the file), then the member function clear must be used before `seekp` or `seekg` is used.

Two other member functions may be used for random file access: `tellp` and `tellg`. They return a long integer that indicates the current byte of the file's read/write position. As expected, `tellp` is used to return the write position of an output file and `tellg` is used to return the read position of an input file.

Assume that a data file letterGrades.txt has the following single line of information:

ABCDEF

Marker positions always begin with 0. The mapping of the characters to their position is as follows:

A B C D E F
0 1 2 3 4 5

The following example program demonstrates the use of `seekg` and `tellg`.

Example Program 14.7

```
#include <fstream >
#include <iostream >
#include <cctype>              // for toupper function



int main ()
{

      std::fstream inFile("letterGrades.txt", std::ios::in);
      long offset;            // used to hold the offset of the read
                                    // position from some point
      char ch;                // holds character read at some
                                    // position in the file
      char more;              // used to indicate if more information
                                    // is to be given

      do
      {
```

```
            std::cout <<"The read position is currently at byte "
                  << inFile.tellg() << std::endl;
            // This prints the current read position (found by the
            // tellg function)

            std::cout << "Enter an offset from the beginning of the file:  ";
            std::cin >> offset;

            inFile.seekg(offset, std::ios::beg);

            // This moves the position from the beginning of the file.
            // offset contians the number of bytes that the read position
            // will be moved from the beginning of the file

            inFile.get(ch);
            // This gets one byte of information from the file

            std::cout << "The character read is " << ch << std::endl;

            std::cout << "If you would like to input another offset enter a Y\n";
            std::cin >> more;

            inFile.clear();
            // This clears the file in case the eof flag was set
      }  while (toupper(more) == 'Y');
      inFile.close();

      return 0;
}
```

Sample Run:

The read position is currently at byte 0
Enter an offset from the beginning of the file: 2
The character read is C
If you would like to input another offset enter a Y
y
The read position is currently at byte 3
Enter an offset from the beginning of the file: 0
The character read is A
If you would like to input another offset enter a Y
y
The read position is currently at byte 1
Enter an offset from the beginning of the file: 5
The character read is F
If you would like to input another offset enter a Y
n

Caution: If you enter an offset that goes beyond the data stored, it prints the previous character offset.

Lab 14 Advanced File Operations

The purpose of this lab is to give the student experience in working with binary files, random access files and other file operations. It begins with a brief quiz followed by a lab that reviews basic file operations.

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. The _____member function moves the read position of a file.

2. Files that will be used for both input and output should be defined as _____data type.

3. The _____member function returns the write position of a file.

4. The ios::_____file access flag indicates that output to the file will be written at the end of the file.

5. _____files are files that do not store data as ASCII characters.

6. The _____member function moves the write position of a file.

7. The _____function can be used to send an entire record or array to a binary file with one statement.

8. The >> operator _____any leading whitespace.

9. The _____function "looks ahead" to determine the next data value in an input file.

10. The _____ and _____functions do not skip leading whitespace characters.

Note: The programs in this lab use the global `using namespace std;` although the lessons did not. You should check with your instructor on the style you should use for the class.

Lab 14.1 Introduction to Files (Optional)

(This is a good exercise for those needing a review of basic file operations)

Retrive program `files.cpp` from the Lab 14 folder. The code is as follows:

```
// This program uses hours, pay rate, state tax and fed tax to determine gross
// and net pay.

#include <fstream>
#include <iostream>
#include <iomanip>
```

```
int main()
{
      // Fill in the code to define payfile as an input file
      float gross;
      float net;
      float hours;
      float payRate;
      float stateTax;
      float fedTax;

      cout << fixed << setprecision(2) << showpoint;
      // FILL IN THE CODE TO OPEN payfile AND ATTACH IT TO THE PHYSICAL FILE
      // NAMED payroll.dat

      // FILL IN CODE TO WRITE A CONDITIONAL STATEMENT TO CHECK IF payfile
      // DOES NOT EXIST.
      {
            cout << "Error opening file. \n";
            cout << "It may not exist where indicated" << endl;
            return 1;
      }

      cout << "Payrate      Hours    Gross Pay       Net Pay"
           << endl   << endl;
     // FILL IN CODE TO PRIME THE READ FOR THE payfile FILE.
     // FILL IN CODE TO WRITE A LOOP CONDITION TO RUN WHILE payfile HAS MORE
     // DATA TO PROCESS.
      {
         payfile >> payRate >> stateTax >> fedTax;
         gross = payRate * hours;
         net = gross - (gross * stateTax) - (gross * fedTax);
         cout << payRate << setw(15) << hours << setw(12) << gross
              << setw(12)  << net << endl;
         payfile >> // FILL IN THE CODE TO FINISH THIS WITH THE APPROPRIATE
                    // VARIABLE TO BE INPUT
      }
      payfile.close();
      return 0;
}
```

Exercise 1: Assume that the data file has `hours, payrate, stateTax` and `fedTax` on one line for each employee. `stateTax` and `fedTax` are given as decimals (5% would be .05). Complete this program by filling in the code (places in bold). NOTE: The program will not run as is. Notice there is no using `namespace std;` Fix the program but do not fix it by including the `using namespace std;` Place `std::` wherever it needs to be placed in the program.

Exercise 2: Run the program. Note: the data file does not exist so you should get the error message:

```
Error opening file.
It may not exist where indicated
```

Exercise 3: Create a data file with the following information:

40 15.00 .05 .12
50 10 .05 .11
60 12.50 .05 .13

Save it in the same folder as the `.cpp` file. What should the data file name be?

Exercise 4: Run the program. Record the output here:

Exercise 5: Change the program so that the output goes to an output file called `pay.out` and run the program. You can use any logical name you wish for the output file.

Lab 14.2 Files as Parameters and Character Data

Retrieve program Grades.cpp and the data file graderoll.dat from the Lab 14 folder. The code is as follows:

```
#include // FILL IN  DIRECTIVE FOR FILES
#include <iostream>
#include <iomanip>
using namespace std;


// This program reads records from a file.  The file contains the
// following: student's name, two test grades and final exam grade.
// It then prints this information to the screen.


const int NAMESIZE = 15;
const int MAXRECORDS = 50;
struct Grades                              // declares a structure
{
      char name[NAMESIZE + 1];
      int test1;
      int test2;
      int final;

};

typedef Grades gradeType[MAXRECORDS];
 // This makes gradeType a data type
 // that holds MAXRECORDS
 // Grades structures.



// FIll IN THE CODE FOR THE PROTOTYPE OF THE FUNCTION ReadIt
// WHERE THE FIRST ARGUMENT IS AN INPUT FILE, THE SECOND IS THE
// ARRAY OF RECORDS, AND THE THIRD WILL HOLD THE NUMBER OF RECORDS
// CURRENTLY IN THE ARRAY.


int main()

{
       ifstream indata;
       indata.open("graderoll.dat");
       int numRecord;                 // number of records read in
       gradeType studentRecord;
```

```
      if(!indata)
      {
            cout << "Error opening file. \n";
            cout << "It may not exist where indicated" << endl;
            return 1;
      }

      // FILL IN THE CODE TO CALL THE FUNCTION ReadIt.

      // output the information
      for (int count = 0; count < numRecord; count++)
      {
         cout << studentRecord[count].name << setw(10)
                << studentRecord[count].test1
                << setw(10) << studentRecord[count].test2;
         cout << setw(10) << studentRecord[count].final << endl;
      }

      return 0;
}

//*************************************************************
//                           readIt
//
//  task:     This procedure reads records into an array of
//            records from an input file and keeps track of the
//                total number of records
//  data in:  data file containing information to be placed in
//            the array
//  data out: an array of records and the number of records
//
//*************************************************************

void readIt(// FILL IN THE CODE FOR THE FORMAL PARAMETERS AND THEIR
            // DATA  TYPES.
           //  inData, gradeRec and total are the formal parameters
           //  total is passed by reference)

{
   total = 0;
     inData.get(gradeRec[total].name, NAMESIZE);
   while (inData)
   {
     // FILL IN THE CODE TO READ test1
     // FILL IN THE CODE TO READ test2
     // FILL IN THE CODE TO READ final

       total++;     // add one to total

     // FILL IN THE CODE TO CONSUME THE END OF LINE
     // FILL IN THE CODE TO READ name

  }

}
```

Exercise 1: Complete the program by filling in the code. (Areas in bold). This problem requires that you study very carefully the code and the data file already written to prepare you to complete the program. Notice that in the data file the names occupy no more than 15 characters. Why?

Exercise 2: Add another field to the record called `letter` which is a character that holds the letter grade of the student. This is based on the average of the grades as follows: `test1` and `test2` are each worth 30% of the grade while `final` is worth 40% of the grade. The letter grade is based on a 10 point spread. The code will have to be expanded to find the average.
90–100 A
80–89 B
70–79 C
60–69 D
0–59 F

Lab 14.3 Binary Files and the write Function

Retrieve program budget.cpp from the Lab 14 folder. The code is as follows:

```
// This program reads in from the keyboard a record of financial information
// consisting of a person's name, income, rent, food cost, utilities and
// miscellaneous expenses.  It then determines the net money
// (income minus all expenses)and places that information in a record
// which is then written to an output file.

#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;

const int NAMESIZE = 15;
struct  budget  //declare a structure to hold name and financial information
{
      char name[NAMESIZE + 1];
      float income;       // person's monthly income
      float rent;         // person's monthly rent
      float food;         // person's monthly food bill
      float utilities;    // person's monthly utility bill
      float miscell;      // person's other bills
      float net;          // person's net money after bills are paid

};

int main()

{
   fstream indata;
   ofstream outdata;                                    // output file of
                                                        // student.
   indata.open("income.dat", ios::out | ios::binary);  // open file as binary
                                                        // output.
   outdata.open("student.out");                         // output file that
```

```
                                                    // we will write student
                                                    // information to.
    outdata << left << fixed << setprecision(2);    // left indicates left
                                                    // justified for fields

    budget person;   //defines person to be a record

    cout << "Enter the following information" << endl;
    cout << "Person's name: ";
    cin.getline(person.name, NAMESIZE);
    cout << "Income :";
    cin >> person.income;


    // FILL IN CODE TO READ IN THE REST OF THE FIELDS:
    // rent, food, utilities AND miscell TO THE person RECORD

    // find the net field
    person.net = // FILL IN CODE TO DETERMINE NET INCOME (income - expenses)

    // write this record to the file
    // Fill IN CODE TO WRITE THE RECORD TO THE FILE indata (one instruction)

    indata.close();

    // FILL IN THE CODE TO REOPEN THE indata FILE, NOW AS AN INPUT FILE.
    // FILL IN THE CODE TO READ THE RECORD FROM indata AND PLACE IT IN THE
    // person RECORD (one instruction)

    // write information to output file
    outdata << setw(20) << "Name" << setw(10) << "Income" << setw(10) << "Rent"
            << setw(10) << "Food" << setw(15) << "Utilities" << setw(15)
            << "Miscellaneous" << setw(10) << "Net Money" << endl << endl;

    // FILL IN CODE TO WRITE INDIVIDUAL FIELD INFORMATION OF THE RECORD TO
    // THE outdata FILE.(several instructions)

     return 0;
}
```

Exercise 1: This program reads in a record with fields `name`, `income`, `rent`, `food`, `utilities`, and `miscellaneous` from the keyboard. The program computes the net (income minus the other files) and stores this in the `net` field. The entire record is then written to a binary file (`indata`). This file is then closed and reopened as an input file. Fill in the code as indicated by the comments in bold.

Sample Run:
Enter the following information
Person's Name: Billy Berry
Income: 2500
Rent: 700
Food: 600
Utilities: 400
Miscellaneous: 500

The program would write the following text lines to the output file `student.out`.

Name	Income	Rent	Food	Utilities	Miscellaneous	Net Money
Billy Berry	**2500.00**	**700.00**	**600.00**	**400.00**	**500.00**	**300.00**

Exercise 2: Alter the program to include more than one record as input. Use an array of records.

Sample Run:

Enter the following information
Person's Name: Billy Berry
Income: 2500
Rent: 700
Food: 600
Utilities: 400
Miscellaneous: 500

Enter a Y if you would like to input more data
Y

Enter the following information
Person's Name: Terry Bounds
Income: 3000
Rent: 750
Food: 650
Utilities: 300
Miscellaneous: 400

Enter a Y if you would like to input more data
n
That's all the information

The output file `student.out` should then have the following lines of text written to it.

Name	Income	Rent	Food	Utilities	Miscellaneous	Net Money
Billy Berry	**2500.00**	**00.00**	**600.00**	**400.00**	**500.00**	**300.00**
Terry Bounds	**3000.00**	**750.00**	**650.00**	**300.00**	**400.00**	**900.00**

Lab 14.4 Random Access Files

Retrieve program `randomAccess.cpp` and the data file proverb.txt from the Lab 14 folder. The code is as follows:

```
#include <iostream>
#include <fstream>
#include <cctype>
using namespace std;
```

```
int main()
{
      fstream inFile("proverb.txt", ios::in);
      long offset;
      char ch;
      char more;

      do
      {
            //  Fill in the code to write to the screen
            //  the current read position (with label)

            cout << "Enter an offset from the current read position: ";
            cin >> offset;

            //  Fill in the code to move the read position "offset" bytes
            //  from the current read position.

            //  Fill in the code to get one byte of information from the file
            //  and place it in the variable "ch".

            cout << "The character read is "  << ch << endl;
            cout << "If you would like to input another offset enter a Y"
                  << endl;

            cin >> more;

            //  Fill in the code to clear the eof flag.

      } while (toupper(more) == 'Y');

      inFile.close();
      return 0;
}
```

Exercise 1: Fill in the code as indicated by the comments in bold. The file proverb.txt contains the following information:

Now Is The Time fOr All GoOd Men to come to the aid of their Family

Sample Run:

The read position is currently at byte 0
Enter an offset from the current read position: 4
The character read is I
If you would like to input another offset enter a Y
y
The read position is currently at byte 5
Enter an offset from the current read position: 2
The character read is T
If you would like to input another offset enter a Y
y

The read position is currently at byte 8
Enter an offset from the current read position: 6
The character read is e
If you would like to input another offset enter a Y
y
The read position is currently at byte 15
Enter an offset from the current read position: 44
The character read is r
If you would like to input another offset enter a Y
y
The read position is currently at byte 60
Enter an offset from the current read position: 8
The character read is r
If you would like to input another offset enter a Y
N

Exercise 2: Why do you think that the character printed at the last run was another 4? What wouldyou have to do to get a different letter after the position is beyond the eof marker?

Exercise 3: Change the program so that the read position is calculated from the end of the file. What type of offsets would you need to enter to get characters from the proverb? Do several sample runs with different numbers to test your program.

Student Generated Code Assignments

Student Generated Code Program 14.5

Write a program that will: 1) read an array of records from the keyboard, 2) store this information to a binary file, 3) read from the binary file back to the array of records, 4) store this information to a textfile. Left justify the information for each field. Each record will consist of the following fields:

first name	15 characters
last name	15 characters
street address	30 characters
city	20 characters
state	5 characters
zip	long integer

You may assume a maximum of 20 records.
This assignment is very similar to the program found in the lesson.

Sample Run:

Enter the following information
Person's First Name: Billy
Person's Last Name: Berry

Street: 205 Main Street
City: Cleveland
State: Tx
Zip: 45679

Enter a Y if you would like to input more data
Y

Enter the following information
Person's First Name: Sally
Person's Last Name: Connely
Street: 348 Wiley Lane
City: San Francisco
State: Md
Zip: 54789

Enter a Y if you would like to input more data
n
That's all the information

The output file contains the following:

First Name	Last Name	Street	City	State	Zip Code
Billy	**Berry**	**205 Main Street**	**Cleveland**	**Tx**	**45679**
Sally	**Connely**	**348 Wiley Lane**	**San Francisco**	**Md**	**54789**

Student Generated Code Program 14.6

Write a program that will read the radii of circles. Use an array of records where each record will have the radius of the circle read from the keyboard and the diameter and area of the circle will be calculated by the program. This information (radius, diameter and area) is stored in a binary file. The information in the binary file is then read back into the records and stored in a text output file. Left justify the information for each field.
Each record will consist of the following fields:

radius	float
diameter	float
area	float

You may assume a maximum of 20 records. You may want to include the `cmath` library.
You need to know the formulas for finding the area and circumference of a circle.

Sample Run:
Enter the following information:
Radius of circle: 5
Enter a Y if you would like to input more data

y

Enter the following information:
Radius of circle: 4
Enter a Y if you would like to input more data
y

Enter the following information:
Radius of circle: 7
Enter a Y if you would like to input more data
n
That's all the information.

The output file contains the following:

Radius	**Area**	**Circumference**
5.00	**78.54**	**31.42**
4.00	**50.27**	**25.13**
7.00	**153.94**	**43.98**

Student Generated Code Program 14.7

Bring in the file `employee.in` from Lab 14 folder. Write a program that will read records from this file and store them in a binary file. That file will then be used as input to create an output file of the information. The data file contains employee information consisting of name, social security, department ID, years employed and salary. In addition to displaying the information of each record the program will also calculate the average salary and years employed of all the records. This additional information is stored in the same output file.

Sample Data File:

Bill Tarpon	182460678	789	8	30600
Fred Caldron	456905434	789	10	40700
Sally Bender	203932239	790	8	50000
David Kemp	568903493	790	9	60000

The output file should look like this:

Name	**Social Security**	**Department ID**	**Years Employed**	**Salary**
Bill Tarpon	**182460678**	**789**	**8**	**30600.00**
Fred Caldron	**456905434**	**789**	**10**	**40700.00**
Sally Bender	**203932239**	**790**	**8**	**50000.00**
David Kemp	**568903493**	**790**	**9**	**60000.00**

The average number of years employed is 8
The average salary is $45325.00

Lesson 14 Summary

Files have an “invisible” end of line marker at the end of each line and an end of file marker at the end of the file. The `eof` marker is an indicator that there is no more data to be read. While loops can use various methods to read data until that eof marker is encounterd.
Example given a data file called infile here are some options:

while (!infile.eof()) while ((ch=infile.peek())!=EOF)

The peek() function looks ahead to see if the next datum is EOF. It does not read it into the program but just “peeks” to see if it is there. That peek is stored in some character variable (ch in the above case).

Input files have the data type `ifstream` while output files have the data ype `ofstream`.

A file can be used for both input and output. The `fstream` data type, which is used for files that can handle both input and output, must have a file access flag as an argument to the `open` function so that the mode, input or output, can be determined. There are several access flags that are used to indicate the use of the file. The following chart lists frequently used access flags.

Flag mode	**Meaning**
`ios::in`	Input mode. The file is used for “reading” information. If the file does not exist, it will not be created.
`ios::out`	Output mode. Information is written to the file. If the file already exists, its contents will be deleted.
`ios::app`	Append mode. If the file exists, its contents are preserved and all output is written to the end of the file. If it does not exist, then the file will be created. Notice how this differs from `ios::out`.
`ios::binary`	Binary mode. Information is written to or read from in pure binary format

Files can be passed as parameters but they need the symbol & after the data type to indicate that they are passed by reference.

Random Access Files means that data can be retrieved from a file in a different order than the order the data was created in the file.
The seekp function is used to go to some position in an output file while the seekg is used to go to some position in an input file.

Example: given some output file dalled dataOut

```
dataOut.seekp(30L, ios::beg);
```

This instruction moves the marker position of the file called `dataOut` to 30 positions from the beginning of the file. The first argument, `30L` (L indicates a long integer), represents an offset (distance) from some point in the file that will be used to move the read/write position. That point in the file is indicated by the second argument (`ios::beg`). This access flag indicates that the offset is calculated from the beginning of the file. The offset can be calculated from the end (`ios::end`) or from the current (`ios::cur`) position in the file.

If the `eof` marker has been set (which means that the position has reached the end of the file), then the member function clear must be used before `seekp` or `seekg` is used.

Two other member functions may be used for random file access: `tellp` and `tellg`. They return a long integer that indicates the current byte of the file's read/write position. As expected, `tellp` is used to return the write position of an output file and `tellg` is used to return the read position of an input file.

APPENDIX A: Case Study

CASE STUDY # 1

Problem: Write an interactive program which, given the number of ages in a list and given the list of ages (integers), finds the lowest (smallest) age in the list.

Clarification: No data editing needs to be done for this program. That is, assume that the user will enter the numbers within the ranges specified below. It will not matter if the list contains duplicate ages.

I**nput**: The first number input will indicate how many ages will follow.
There will be from 1 to 10 ages.
An age will be in the range 0 to 125, inclusive
The user will be prompted for each age.
There will be at least one age listed.

Output: Write a brief opening message (shown only for the first set of sample session below for the sake of space), explaining what the programd does.
Prompt the user to enter the number of ages and, in turn, each age.
Display the smallest age, labeled, after all ages have been read in.

Sample Input/Output

Sample Session #1, with output in bold:

This program finds the smallest age from a list of ages which you will enter at the keyboard. The ages must be in the range 0 to 125, inclusive.

Number of ages >2
Age > 12
Age > 9

Smallest age = 9

Sample Session #2

This program finds the smallest age from a list of ages which you will enter at the keyboard. The ages must be in the range 0 to 125, inclusive.

Number of ages > 3
Age > 68
Age > 68
Age > 68

Smallest age = 68

Sample Session #3:

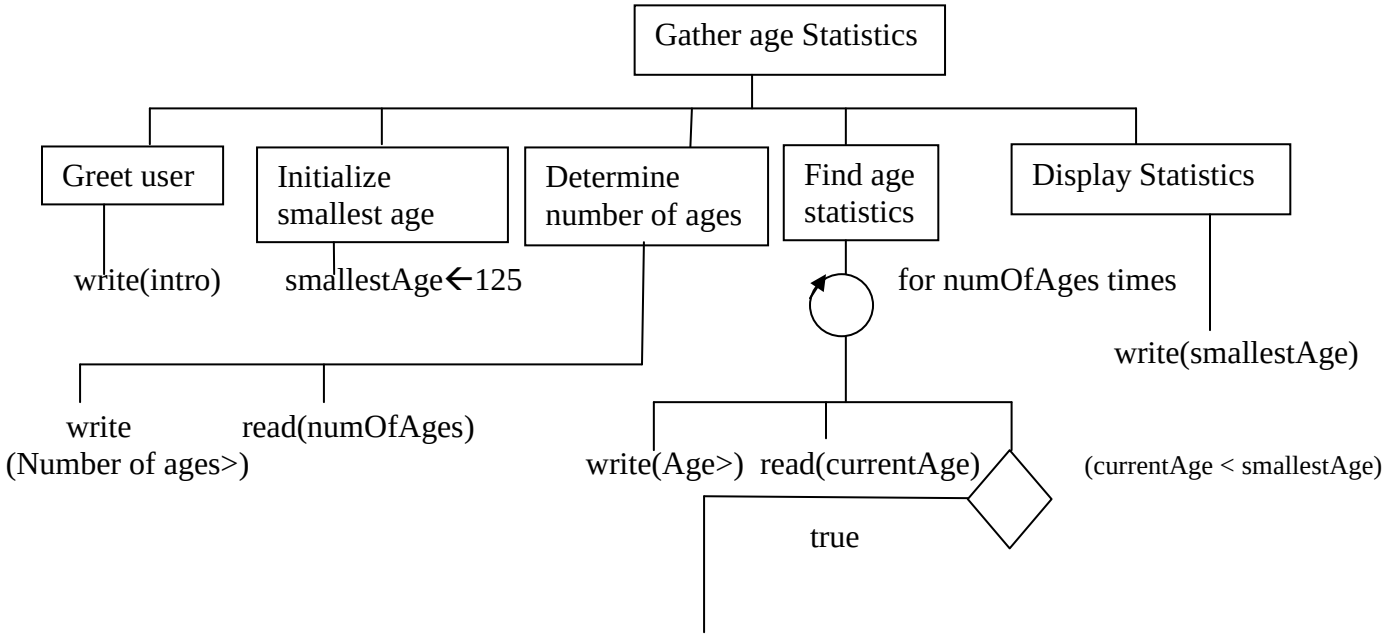
Number of ages> 9
Age >18
Age >49
Age >65
Age >17
Age >22
Age >17
Age >81
Age >17
Age >17

Smallest age=17

DESIGN for Ages Program

DATA:	**Name**	**Type**	**Description**
Input Variables:	numOfAges	integer	number of ages to be read in
	currentAge	integer	age just read in
Output Variables:	smallestAge	integer	smallest age from among those read in
Other Variables:	counter	integer	loop counter

ALGORITHM (tree structure chart)



The SAME ALGORITHM in PSEUDOCODE REPRESENTATION with comments in parentheses:

```
(Greet user)
write out message to user telling what program does

(Initialize)
smallestAge <---125

(Find out how many ages to be entered)
write out “Number of ages ?”
Read in numOfAges

(Find age statistics)
Loop for counter going from 1 to numOfAges:
        prompt for next age
        read in currentAge

        (Examine to see if stats need to be updated)
        if (currentAge < smallestAge)
                smallestAge <----currentAge

(Display statistics)
write out”Smallest age is”, smallestAge
```

C++ Program

```
//_____________________________________________________
//Program:      Smallest Age
//Programmer: Name
//
//This program reads in a list of ages from the keyboard and
//then displays the smallest age in the list
//-----------------------------------------------

#include <iostream>
using namespace std;

void main()
{
        int     numOfAges,   //number of ages to be entered
                currentAge,    //age currently being looked at
                smallestAge;  //smallest age in list

        //Greet user
        cout<< "\nThis program finds the smallest age from a list of ages\n"
           << "which you will enter at the keyboard.  The ages\n"
           << "must be in the range 0 to 125, inclusive.\n\n";

        // Initialize (you may initialize in declaration
        smallestAge=125;

        //Find out how many ages to be entered
        cout << "Number of ages?";
        cin >> numOfAges;

        //Find age statistics
        for (int counter=1; counter <= numOfAges; counter++)
        {
                cout << "\nAge?";
                cin >> currentAge;

                //Examine to see if stats need to be updated
                if (currentAge < smallestAge)   //this age smaller than smallest
                        SmallestAge=currentAge;
        }
        //Display statistics
        cout << "\n\n"
            << "The smallest age is "   << smallestAge;
}
```

Appendix B: Case Study using Functions

DESCRIPTION: Develop a program that will read characters representing binary(base 2) numbers from a data file and translates them to decimal (base 10) numbers. The decimal numbers should be output in a column with an appropriate heading. Each line of the file will contain a number that indicates the size of the binary number (how many bits) followed by the binary number with each digit separated by one or more spaces. The program reads those digits one at a time. As each digit is read, the program should translate that digit into the corresponding decimal value by multiplying it by the appropriate power of 2 (depending on where the digit was in the number. The program should print the decimal sum of all the numbers in the file and give the average of all those numbers.

CLARIFICATION: There is only one decimal number (indicating the size of the binary number) and one binary number following it per line. There is an arbitrary number of blanks before each number and each digit of binary number. The program should process all the lines in the file. Use plenty of comments, proper documentation and coding style, and meaningful identifiers throughout the program. You must use at least 2 procedures (functions) besides the main program. You must decide which of your design modules should be coded as functions (procedures) to make the program easier to understand. You should use some of the built in math functions. (So include #include <cmath> library routine.) For full credit it should check for bad data; if it encounters anything except a zero or a one, it should output the message "Bad integer on input", and will not process that particular number any further. The average should have a precision of 2.

Conversion of a binary number to decimal:

Binary numbers consist of only 1's and 0's.

The place value of a binary number is as follows

32	16	8	4	2	1

Note that each position is twice the value of its neighbor to the right.

Example: 1 1 0 1 binary equals

1	1	0	1
8	4	2	1

1 *8 + 1 *4 + 0*2 +1* 1 = 13

Note whenever a 0 in encounter nothing is added to the value and whenever a 1 is added the value of that place is added to the total value of the number.

SAMPLE INPUT:

```
5  1 0 1 1 0
4  1 1 9 0
7  1 1 1 0 1 0 1
10 1 0 1 1 1 0 0 0 1 0
```

SAMPLE OUTPUT:

The decimal value of the binary number 10110 is 22.

The decimal value of the binary number 11 9 is Invalid binary number.

The decimal value of the binary number 1110101 is 117.

The decimal value of the binary number 1 0 1 1 1 0 0 0 1 0 is 738.

The Sum of the valid numbers is 877.

The average of the valid numbers is 292.33.

List of Variables

Input Variables

digitamount	int-----> the number of bits in the binary number
bit	int----->the individual bit 1 or 0 for valid number

Output Variables

decimalvalue	int---->the value of the binary number in decimal
totalsum	int---->the total
average	float--->the average of all the number in the file in decimal

Intermediate Variables

valid	bool---->check for valid binary digit 1 or 0
amount0fNum	int------->the amount of valid numbers in the file

Algorithm

// Convert binary to Decimal

//initialize data

totalsum<-------0;
amountOfNum<-----0:

while (NOT EOF)

// Include other instructions here

```
   // process digits
        for counter going from digitamount downto 1
                read (bit)
                // check if bit is valid

                write(bit)
                If (CHECKBIT(bit)      // Call to a  function
                        CALL PLACEVALUE(bit, digitamount, decimalvalue)    // call to a procedure
                             // Include other instructions here
                else
                        // Include other instructions here

     // add amount to sum and increment the amount of valid numbers
          If (valid)
                // Include proper instructions here

//Write out summary report
        CALL SUMMARY (totalsum, amountOfNum)

//end of main
```

PROCEDURE Check Bit

Task Checks to see if bit read is a one or 0. If so then it is valid
Otherwise it is invalid

Data in: bit An integer

Data out: valid Boolean True if integer is 1 or 0 False otherwise

```
        // body of function Include instructions here
```

PROCEDURE PLACEVALUE

Task: To take the current bit and figure its value based on the current degitamount and add that amount to the total value of the number thus far.

Data in: bit integer current value of the bit (assumes bit is valid)
Decimalvalue Integer that has the current value of the number so far
Digitamount integer of the place value of the current bit

data out: decimalvalue Has been altered.

// body of procedure Include instructions here

PROCEDURESUMMARY

Task:	To print out the totalsum of all the valid numbers and the average of those same numbers
data in:	totalsum Integer Sum of all valid numbers amountofnum integer the number of valid numbers
data out	average

// Body of procedure include instructions here

Appendix C: Reserved Words and Identifiers.

A. Reserved Words--These are words that have special meaning in C and they cannot be used for any other purpose. They will appear in lowercase. They are words such as int, double, return, void, int.

B. Standard identifiers--These are words that have special meaning in C++ libraries. They are defined in programs called libraries in the C++ compiler. Although they can be used for other purposes DO NOT use them for anything else.

C. Object declarations--user defined words that will be used to give names to memory locations. Object declarations names are identifiers. The values stored there can change during the course of the program.

D. Identifiers--Identifiers are used for object declarations as well as many other uses.

1) an identifier cannot begin with a digit
2) an identifier must consist only of letters, digits or underscore
3) C reserved word cannot be used as an identifier and standard identifiers should not be redefined.

E. Constants-- Constants by const datatype nameofconstant = value of constant. This allocates a memory location and places the value given it by the directive in that location. It will NOT change at all.
const int sum=1;
const float pi=3.14;

Uppercase and Lowercase letters

NOTE: C++ compiler considers the use of uppercase and lowercase as significant. The names Rate, rate, and RATE are all viewed by the compiler as different identifiers. We use all caps for constants (such as KMS_PER_MILE) and all lowercase for objects, reserved words and standard identifiers.

List of Reserved Words in C++

auto	default goto	friend	public	this
break	do	if	register	template
case	double	inline	return	typedef
catch	else	int	short	union
char	enum	long	signed	unsigned
class	extern	new	sizeof	virtual
const	float	overload	static	void
continue	for	private	struct	volatile
delete		protected	switch	while

Appendix D: Input and Output Statements

1) Input operations

Objects can be stored in memory locations through the use of a function like cin.

All input and output operations in C++ are performed by executing special program units called input/output functions. The most common functions are found in the stream and fstream input/output library.

#include <stream> includes that library so that the program can have access to cin and cout.

cin >> miles;

Input from keyboard

#include <stream>;

Input from data file

#include <fstream>;

ifsteam infile; // infile is name of data file used

infile.open("pay.dat");

Then we can use the word infile.

infile >> miles >> distance;

2) Output operations

Output operations get information out so that humans can read what the computer has calculated.

cout is a function that performs such a task.

cout << "That equals " << kms <<" kilometers. \n";

formating output

```
cout.width(10);  //set minimum field width to 10
cout.fill('%');  // set fill character
cout.setf(ios::left);  // left justify
cout.precision(3);  // set floating point precision to 3
```

Note: cout could be substituted by any output file properly defined.

```
ofstream outfile;
outfile.open("pay.rpt");

outfile.width(10);

etc.
```

\ is a special symbol that gives directives to follow. It is followed by another symbol which indicates the action. Both are inclosed in double quotes.

\n states that the output should go to the next line after executing the function.
endl also goes to next line << endl; or << "\n";
\r return
\b backspace
\t horizontal tab
\f will advance to next page but won't show on screen

3) Formatted output Manipulators

A. setiosflags is used to change default format state flags. These can be stand alone statements or part of a cout statement.

1. setiosflags(ios::left); //output left justified
2. setiosflags(ios::right); //output right justified
3. setiosflags(ios::showpoint);
   // This causes a floating-point value to be printed with decimal point
4. setiosflags(ios::fixed);
   //Put all values in conventional notation.
5. setiosflags(ios::scientific);
   //output placed in scientific notation.
6. setiosflags(showpos); // prints numeric values with a + sign.

B. resetiosflags is used to reset or turn off the flags specified.

1. resetiosflags(ios::left);
   //Turns off left justification

C. Examples:

```
cout << endl << setw(10) << setfill(' *') << fixed <<  number;
```

Appendix E: Shortcut Statements of C++

object++ adds one to the value of the object
object-- subtracts one to the value of the object
++ and --

1) y= x++ y gets the value of x then x is incremented by 1.

2) y= ++x x is first incremented then y gets the new value of x

3) y= x-- y get value of x, then x is decremented

4) y=--x x is decremented then y gets the new value of x

5) y++ is a shortcut for y= y+1; y—is a shortcut for y=y-1;

(((example y = y+1 can now be written y++)))

If x = 3 what would value of y and x be after each of the above statements were executed.

y = x++	y=3	x=4
y= ++x	y=4	x=4
y= x--	y=3	x=2
y= --x	y=2	x=2

ANOTHER shortcut;

```
sum += total;     // this will add the contents of total to sum and
store                          // it in sum
```

it is the same as sum = sum + total;

Appendix F: Math functions

A list of mathematical functions in the <cmath> library.

sin(x)	sine of x
cos(x)	cosine of x
tan(x)	tangent of x
asin(x)	$\sin^{-1}(x)$
acos(x)	$\cos^{-1}(x)$
atan(x)	$\tan^{-1}(x)$
atan2(y,x)	$\tan^{-1}(y/x)$
sinh(x)	hyperbolic sine of x
cosh(x)	hyperbolic cosine of x
tanh(x)	hyperbolic tangent of x
exp(x)	exponential function e^x
log(x)	natural logarithm ln(x)
log10(x)	base 10 log
pow(x,y)	**x^y**
sqrt(x)	**square root of x**
ceil(x)	**smallest integer not less than x**
floor(x)	**largest integer not greater than x**
fabs(x)	**absolute value of x**

Appendix G: String Functions

Needs library **string**

strcat (string1,string2)	Concatenates string2 to string1
strchr(string,character)	Locates the position of the first occurrence of the character within the string. Returns the address of the character.
strcmp(string1,string2)	Compares string1 to string2. Returns a negative integer if string1 < string2, 0 if string1 ==string2, and a positive integer if string1 > string2.
strncmp(string1,string2,n)	Compare at most n characters of string1 to string2. Returns the same values as strcmp() based on the number of characters compared.
strcpy(string1,string2)	Copies string2 to string1, including the '\0'. Returns string1.
strncpy(string1,string2,n)	Copies at most n characters of string2 to string1. If string2 has fewer than n characters it will pad string1 with '/0's. Returns string1.
strlen(string)	Returns the length of the string.

Needs library **stdlib.h**

atoi(string)	Converts an ASCII string to an integer.
atof(string)	Converts an ASCII string to a real.
itoa(string)	Converts an integer to an ASCII string.

Needs library **cctype**

isalpha(character)	Returns a nonzero number if the character is a letter; otherwise it returns a zero.
isupper(character)	Returns a nonzero number if the character is uppercase; otherwise it returns a zero.
islower(character)	Returns a nonzero number if the character is lowercase; otherwise it returns a zero.
isdigit(character)	Returns a nonzero number if the character is a digit (0 through 9); otherwise it returns a zero.
toupper(character)	Returns the uppercase equivalent if the character is lowercase; otherwise it returns the character unchanged.
tolower(character)	Returns the lowercase equivalent if the character is lowercase; otherwise it returns the character unchanged.

Appendix H: Escape Sequence

Escape Sequence	Name	Meaning
\a	Alert	Sounds a beep
\b	Backspace	Backs up one character
\f	Formfeed	Starts a new screen or page
\n	Newline	Moves to beginning of next line
\r	Carriage return	Moves to beginning of current line
\t	Horizontal tab	Moves to next tab position
\v	Vertical tab	Moves down a fixed amount
\\	Backslash	Prints a backslash
\’	Single quotation	Prints a single quotation mark
\”	Double quotation	Prints double quotation marks
\?	Question mark	Prints a question mark
\000		Prints a character whose ASCII code is a one-to-three-digit octal value
\XHHH		Prints a character whose ASCII code is a one-to-three-digit hexadecimal value

<ctrl> P L Goes to a new page in the source program

system(“cls”); // to clear the screen

Appendix I
Random Number Generators

Random numbers are used to in any program that needs to simulate chance operations. Certain games,(those involving rolling dice or shuffling cards in which an outcome is based on chance) and natural occurrences (where simulation of nature often involves chance) use random functions.

The cstdlib C++ standard library provides two functions for producing random numbers.

The first function has the prototype: int rand();
The second function has the prototype: void srand(unsigned int);

This returns a random integer in the range of 0 through RAND_MAX, (a constant defined in cstdlib). The programmer can generate any range of numbers by being creative with this function.

Example: Generating random numbers from 1 to 52 (cards in a deck) could be done as the following:
card=rand()% 52 +1

Random number generators use an initial seed value from which to start the sequence of random numbers. The seed is initially one. The programmer can select a seed by using another function in C++ called srand. void srand(unsigned int);

The problem with having the same seed is that is will repeat a sequence of random generated numbers.

We can generate a different seed each run by getting that seed from the system clock

Getting a random seed from the system clock

#include <ctime> **//** This library includes the system clock variable time
#include <cstdlib> // This library contains the random number generator functions
srand(time(NULL)); **//**This generates a seed from the system clock

roll = rand()%6 +1; // This will generate a random number from 1 to 6
card = rand()%52 +1; // This generates a random number from 1 to 52 (for use in a deck of cards)

Creating a random number generator as a class:

```
// Specification file (rand2.h)

class RandGen
{
 public:
        float NextRand();
                //POST: FCTVAL==next pseudorandom number in range of
```

```
                    // 0.0 < FCTVAL < 1.0
// constructors:

        RandGen(long initSeed);
         //POST: Pseudorandom sequence initialized using initSeed

        RandGen();
        //POST: Pseudorandom sequence initialized using a default
        //      initial seed
 private:
        long currentSeed;
};
```

Sample client code using two separate random number generators, one for rolling dice and the other for playing cards.

```
#include “rand2.h”

int die1;
int die2;
int playingCard;

RandGen diceGen(359); //Initialize using first constructor

die1= int(diceGen.NextRand()*6.0) +1;
die2= int(diceGen.NextRand()*6.0) +1;


RandGen cardGen;   //Initialize using default constructor

playingCard=int(cardGen.NextRand()*52.0) +1;
```

The implementation of these functions are reserved for a the next programming course.

Appendix J
Debugger

Using the Visual C++.NET Debugger

A debugger allows you to view the values of your variables in a "watch window" as the program is executed. It also allows you to step through your program one instruction at a time, which lets you watch your variables as they change. A debugger is thus useful for finding logic errors in programs. It does a trace of variables which help pinpoint where the logic problems maybe hiding. A program must compile successfully before the debugger can be used. The debugger is most commonly used to step through the lines of code on a line-by-line basis, set variable watches to view the values of individual variables at execution time and set breakpoints (lines of code selected by the programmer at which execution is halted so the programmer can examine the code and the current state (such as variable values) of the program's execution.

These notes give you a brief introduction to the Visual C++ debugger. If you use another programming environment, you will likely have a debugger at your disposal, and it will likely have the same features outlined here. Learning how to use a debugger, regardless of the language which you use, is an important part of programming today. The first three pages are just pre-lab reading. (Pages 153-155).

The debugger will not work properly if your program has syntax errors and/or doesn't link properly.

I. Starting a Debugging Session.

In Visual C++.NET open your C++ program in a window and be sure that it compiles and links properly.
You start a debug session by selecting **DEBUG------->Step Into**
This gets you into a debugging session and will allow the debug commands to respond.
Continue F5---- Once started, this runs the debugger up until the first breakpoint or until the program waits for input.
Step Into F11 This will step through the program one instruction at a time and will enter a function

II. Using Breakpoints

Breakpoints are an important part of debugging. They're like having a stop sign embedded in your program. When your program encounters one, it stops execution and waits.

Why use breakpoints? If you have localized the part of your program which is causing problems, you can set a breakpoint at the beginning of that chuck of code. Then you can run the program to the breakpoint and then trace from there.

Breakpoints are set by clicking the line in the program where the breakpoint is desired and then clicking the right button on the mouse and then selecting **Insert Breakpoint**. (The icon that looks like a hand).

When a breakpoint is set, a small red dot appears on the left of the line. An individual breakpoint can be removed by clicking the line with the breakpoint and clicking the right button on the mouse and selecting **Remove Breakpoint** on the menu.

III. Watch Windows

The bottom portion of the window during a debug session has a window that has three options to show variables: Auto, local and Watch 1

The **Auto** tab displays the name and value of the variables or objects used in both the previous statement and the current statement. The **Locals** tab displays the name and current value for all the local variables or objects in the current function's scope. The **watch 1** window allows the user to watch variables. When the value of a particular variable changes, the user can immediately sees the new value in the **Watch**1, **Locals** or **Autos** Variable pane.

Variables can by typed directly into the **Watch** window or a variable can be dragged from the program into the **Watch** window. A variable can be deleted from the Watch window by highlighting the row containing the variable name and pressing the Delete key.

IV. Debug toolbar and menu

The debug toolbar and menu contains buttons that facilitate the debugging process.

The **Restart** (<ctrl> <shift> <F5>) button restarts the program. Control stops at the first breakpoint or executable line requiring input.

The **Stop Debugging** (<shift><F5>)button ends the debugging session.

The **Step Over** (F10) button executes the next executable line of code and advances the yellow arrow to the next executable line. If the line of code contains a function call, the function is not entered (step into) rather it is executed in its entirety. This process allows the user to execute the program on a line-by-line basis.

The **Step Into** (F11) button allows functions to be stepped into–such that the user can confirm the proper execution of the function line-by-line. This includes programmer defined functions and C++ library functions.

The **Step Out** (<shift><F11>) button allows the user to step out of the current function and return control back to the line of the function call.

There are many other options to be used with a debugger, but this should get you started.

DEBUGGER LAB

Copy the debugger folder (K drive—>Henson—>COSC—> COSC 120—>Student Labs) onto the C drive.

Debugger Lab 1: Introduction to the Debugger

1) Get in to Visual C++.Net and create a project called LabD and bring in debugger.cpp and compile it. Now try to run it by putting in 120 for miles and 2 for hours. Record your result.

2) Something is wrong. We have a logical error somewhere. We can use the debugger to help us find the error.

3) From the **Debug** menu select **Step Into.**

4) Follow the instructor's example to find out how this problem is solved.
F10 ----- steps over functions
F10
F11 ---- steps into the procedure to get the data
F10
F10
F10 Input the data
When you have solved the logic error you can stop debugging (<shift> F5) NOTE: continue to select F10 until the program is done if you want to see the rest of the program executed one instruction at a time.

5) NOTE: You can copy variables in the Watch pane by highlighting the variable and then dragging the box to the Watch pane window.

6) NOTE: Remember that F10 steps over functions and should be used when executing the program one instruction at a time EXCEPT when there is a call to a user defined function that you want to step into. In that case use F11.

If you ever get into "strange code" you can hit < shift> F11 (perhaps more than once) to get out of that code. (That is the step out operation). You can also get out of "strange" code by selecting **close** from the **File menu**.

Debugger Lab 2: Basic Debugger Operations

1) If you are still in the debuglab project, remove debugger.cpp from the project. (If you are not in the debuglab project, open the project or create a new project for the rest of the labs.

2) Add count1.cpp to the project. It is in the debugger folder that you copied earlier. Double Clip count1.cpp to bring it in the window.

3) Now build count1.cpp. You should get no syntax errors.

4) From the Debug menu, select **Step Into** (F11).

5) You should see a pane (this pane will be referred to as the variable pane) in the lower left corner that is labeled either autos, locals or watch 1. [1] These are the three options that can be chosen for this window. (At the bottom of this window you will see the three tabs: autos, locals watch1 which allow you to pick the window you want to display). The autos window displays the contents of the most recently used variables. The locals window displays the contents of the local variables and the watch1 is the window that displays the contents of the variables chosen by the user.

6) By selecting **Step Into** in Step 4, we told the debugger to take one step into our program. That means that the debugger will go to the very beginning of the program and stop. In this exercise we will "single step" through the program. Single stepping means that we work through the program executing one statement at a time checking memory contents after each statement is executed.

7) In the source code page a yellow arrow appears in the left margin. This arrow always points to the next statement that will be executed. Since the arrow currently points to the opening brace of the main function, it indicates that no statements have been executed yet and the program is stopped temporarily at the very beginning of the program.

8) The variables pane at the bottom left is blank (no matter which option is chosen). This tells us that at this point in the program (before any definition of variables) no variables exist.

9) Tell the debugger to execute the next statement in the program. This can be done by selecting the **Step Over** option from the debug menu or by pressing the **F10** key.

10) The yellow instruction pointer arrow jumps down past the variable definitions to the first executable statement in the program: the assignment statement that places 1 into the variable counter. The debugger handles all variable definitions as a single step. The initial values of all variables defined are displayed in the variable pane (option autos or locals). The variable pane effectively allows us to look into the computer's memory and see at any point in time exactly what value is stored in any variable.

11) Since we have not yet explicitly assigned any values to the variables in our program, the variables contain arbitrary values. However, even though the values in the variables are meaningless to us, they are perfectly valid to the program and any operations that the program might perform involving these variables will be executed as if the values they contain are legitimate. Note that although the program has stopped at the statement counter = 1;, that statement has not executed. Each time the debugger stops at a line in the program, it stops before executing the statement on that line.

12) Execute the assignment statement in the program using one of the methods described in Step 9 to invoke the **Step Over** command. The assignment will execute and the value of counter

[1] If you don't see this pane, you should see an icon called Autos near the bottom of the screen. Click on this Autos icon.

displayed in the Variables pane will change to reflect the assignment. (New changes are indicated in red in the variables pane.)

13) Execute the next statement in the program using the **Step Over** operation. Verify that the value of the variable sum changes to reflect the assignment.

14) At this point, you may notice that the variable dataValue disappears from the autos option (but it is maintained in the locals option) of the Variables pane. Since a program might contain many variables, the debugger attempts to anticipate your needs and displays only those variables it thinks are most important at this point in the program. To see all the variables, at once, select the **Locals** tab in the Variables pane.

15) JUST READ THIS STEP! DO NOT TAKE ANY ACTION!!
The debugger can not step backwards and undo the execution of a statement it has already passed but it is possible to restart debugging a program from the beginning at any time by selecting **Restart** from the **Debu**g menu. Restarting clears everything out of the program's memory and resets the program to its initial state. You can also stop the debugger completely at any time by selecting **Stop Debugging** from the **Debug** menu.

16) The program is now stopped at the while statement. A glance at the value of counter in the variable pane shows that the value of counter, 1, is surely less than or equal to 4 so the while condition should evaluate to true and the program should continue execution at the first statement inside the body of the while loop.

17) Execute the next statement and verify that the yellow instruction pointer does indeed move down to the cout statement in the loop body.

18) Invoke **Step Over** again to make the output statement execute. To see the output generated by the program you must bring the console window into focus by clicking on it directly. This window should have an icon that is black (with a blue bar at the top) with the name of the path of the program on it.

19) Execute the cin input statement. Note that the yellow instruction pointer does not move to the next line in the file. This is because the extraction operation is not complete - you must enter the data value requested by the program. Bring the console window into focus (if it does not pop up). Type an integer and then press Enter.

20) Notice that the yellow instruction pointer has moved down to the statement immediately following the input statement just completed. You should also verify that the number you entered is correctly stored in the variable dataValue by checking the Variables pane.

21) Execute the next two statements in the body of the loop and verify that values of sum and counter are properly updated. At this point the yellow instruction pointer is pointing to the closing brace of the loop body. Invoke **Step Over** again to move the instruction pointer back to the top of the loop where it will re-evaluate the loop condition.

22) Continue to single step through the program, entering values when prompted and verifying that all variables are properly updated as the loop executes. Eventually counter will reach a value of 5 and the loop condition will fail. The yellow instruction pointer will then jump past the loop to the output statement immediately following the loop body.

23) Continue to step through the program until the yellow instruction pointer points to the closing brace of main. Select **Stop Debugging** from the Debug menu. If you continue to step through the program you will end up looking inside special VC++.net generated code that is not part of your program. The **Stop Debugging** command will exit the debugger at any time.

24) Stepping through a program one statement at a time is powerful, but tedious. It is also impractical for large programs. When debugging a program, errors usually only occur in a few locations. Most of the program is correct and we don't want to waste time stepping through statements that we know are working correctly. What we'd like to do is skip over the parts of the program that work correctly and then stop at and slowly step through the parts of the program that are causing problems. There are several ways of doing this in the Visual C++.NET debugger. We look at only the use of breakpoints here.

25) Setting a breakpoint in a program is like putting up a stop sign on a highway. Once we have set a breakpoint, we reset the debugger to the beginning of the program and tell it to start running. The debugger will then run our program at full speed all the way up to the statement where we have set the breakpoint. The debugger than halts execution of our program and waits for us to enter additional debugging commands. Once the debugger is stopped at the breakpoint, we can start stepping through our program one line at a time. This process is shown in the following steps.

26) Start the debugger by selecting **Step Into** from the **Build** menu. The yellow arrow appears in the source code pane of the debugger indicating that our program is poised at the beginning.

27) Assume that we are only interested in checking that the sum is accumulating correctly inside the body of the while loop. We set a breakpont at the statement that accumulates the sum by moving the cursor in the count1.cpp source code file to the statement sum = sum + dataValue. Right click the mouse to get the debug menu and select **Insert Breakpoint**.

28) When the breakpoint is set, a small red stop sign icon will appear in the left margin of the source code pane. It can be removed by selecting **Remove Breakpoint** or be disabled by selecting **Disable Breakpoint** from the menu.

29) Continue to run the program by selecting **Continue(F5)** from the **Debug menu.**

30) Type a value and press Enter. Bring the debugger window back into focus if it does not do so automatically. The yellow arrow is now superimposed on the red stop sign icon in the source code window. This indicates that the debugger has stopped our program at the statement where we set the breakpoint.

31) We can now step through the program one line at a time to verify the correctness of the sum accumulation. By looking at the Variables pane, we can see the current values of sum and dataValue and predict what should appear in sum after the current statement is executed.

32) Execute the current statement by invoking the **Step Over** command and verify that sum was correctly updated. At this point we can select **continue** again to have the program resume running at full speed while it reads the next value before it stops again at the breakpoint. We can set and clear breakpoints at any time during a debugging session and it is possible to have multiple breakpoints set at any time.

33) End the program by selecting Stop Debugging from the Debug menu.

Debugger Lab 4: Using the Debugger with Arrays and Functions

1) If you are still in the debuglab project, remove count1.cpp from the project. (If you are not in the debuglab project, open the project or create a new project for this lab.

2) Save both Arrays.cpp and Arrays.dat to the project folder and then add them both to the project. Both of these are found in the debugger folder that you copied at the beginning of the lab. Arrays.cpp reads values into an array from the file Arrays.dat, computes the absolute value of all elements in the array, and then prints the modified values to the screen. This lab will demonstrate several debugger features while illustrating basic array operations.

3) Build Arrays.cpp.

4) Run the program and record the output.

5) Open the file Arrays.dat (click this file from the solution explorer window) and record the contents of the file.

6) The program converted all the negative values in the file to positive values before printing them out. The remainder of this lab will show step-by-step how this was done.

7) Close the file Arrays.dat.

8) In Arrays.cpp set a breakpoint at the statement infile.open("Arrays.dat");. From the **Debug menu** select **Step Into**. Then select **Continue** (F5) from the **Debug menu**.

9) Examine the Variables pane (local or auto) and record the values displayed for the following variables:
index _____

MAX_ARRAY _____

numbers[MAX_ARRAY] _____

10) The variable index contains the expected uninitialized values and the constant MAX_ARRAY contains 5 as expected. infile is not a simple variable- it is an object. An object is a special data type that is composed of many parts–similar to an array or structure but even more complex. If you're interested, you can see the individual pieces by clicking on the '+' just to the left of infile. This expands the display of an object in the same manner as you can expand the display of folders in Windows.

11) Another way to display numbers is to add the array to the Watch1. (One of the options on the Variable pane.) **Select the Watch option.** Immediately under the name column header in the Watch pane, enter the name numbers. Expand the display to show all the elements of the array by clicking on the '+'. Note that the identifier numbers itself contains an address (or reference) rather than an actual value in the array.

12) Clear the breakpoint set at the infile.open("Arrays.dat"); statement. This can be done by moving to that line, clicking the right button on the mouse and selecting **Remove Breakpoint**.

13) Set three new breakpoints at the following lines:
inFile >> numbers[index]; (inside the first for loop)
numbers[index] = abs(numbers[index]);
(inside the second for loop)
cout << arr[i] << ' '; (inside the PrintArray function)

14) Select **continue (F5)** to execute the program up to the first breakpoint. Click on either Autos or Locals for the variable pane. Note that in the Variables pane, the variable index displays the value 0 in red. Remember that red indicates that the variable's value was changed by the last statement executed. In this case, it was changed by the initialization expression associated with the for loop. The variable index indicates the position in the array where the next value read will be stored. Each **time continue (F5)** is clicked the statement inFile >> numbers[index]; is executed. Click **continue** repeatedly and observe how each individual element of the array changes as a new value is read in from the file and placed in the next position of the array. Stop when the program reaches the breakpoint at numbers[index] = abs(numbers[index]);. Record the element values displayed in the Watch pane.

numbers[0] _____
numbers[1] _____
numbers[2] _____
numbers[3] _____
numbers[4] _____

15) The second loop computes the absolute value of each element by calling the standard library function abs(). The result of each function call is stored back in the corresponding element of

the array. Invoke the **Step Over (F10)** operation four times and watch as the program steps over the abs() function to compute the absolute values of the first two elements in the array. Be selecting the Step Over operation, the debugger executes the abs() function all at once rather than one statement at a time. We might assume that since the abs() function is part of the standard libraries, it must be correct and we need not look inside it. However, if we like, we can look inside any function, either library or our own, by invoking the **Step Into (F11)** operation.

16) Remember if you find yourself in an unfamiliar piece of source code while debugging (such as inside the abs() library function), you can usually get back to familiar territory by invoking **Step Out (Shift+F11)**.

17) Click **continue** repeatedly and observe how the elements in the array appear one at a time. You may have to switch back and forth between the output window and the window containing the program.

18) Continue to experiment with the debugger. When you are finished you should **stop debugging** (from the **Debug Menu**). You then may close the project, exit C++ and logout.

Appendix K: Design & Program Rules & Grading

Here are suggestions for guidelines used in programming assignments. Your instructor may give you a different set then those outlined here.

COSC 120 / GUIDELINES FOR PROGRAMMING ASSIGNMENTS /

Part One: PROGRAM DESIGN (as shown in class)

This is due at the beginning of class -- IN DUPLICATE -- one week after receiving the programming assignment . The sections (listed below) must be in order, with each part beginning on a new page. The design must be placed in the pocket of a folder which is clearly labeled on the front with your name, course, and section number. The design should be LANGUAGE INDEPENDENT. Points will be deducted for C++ instructions or notation used in the design.

Include --

1. An UPDATED copy of the problem specifications.
   That is, if the specs have been modified since they were distributed,
   any changes must be reflected in the copy submitted.

2. At least one new set of SAMPLE I/O to show your instructor that you understand the problem.
   Your sample I/O must be sufficient to show your understanding of any special cases.
   Label the data "Sample Input" and "Sample Output".

3. A list of variables, organized in three sections:
   INPUT VARIABLES totalWeight float Accumulates weight of hogs
   OUTPUT VARIABLES
   OTHER VARIABLES.

   Each variable should be listed by NAME, DATA TYPE, and a brief description. For input data which is "echoed" to output it is sufficient to write "Echoed" next to it in the list of output variables.

4. The STRUCTURE CHART ("solution tree"), neatly drawn.

   Nodes must be descriptive. Each must begin with a VERB, as in "Calculate average". Each leaf will be either fundamental instruction or a subroutine call (not for 1st program), or will represent some trivial task (e.g., "Print error message"). You need not refine "trivial tasks" further. Actual messages, prompts, headings, etc need not be spelled out in the chart.

5. The algorithm (using fundamental instructions) derived from the structure chart, showing fundamental instructions on the left and comments on the right.

   Be sure to hand trace your algorithm from the structure chart to confirm that it is correct

before writing the algorithm.

Your grade on your DESIGN package will be based on the following:

i) how well you followed directions;
ii) completeness of the package; and
iii) the extent to which your design is correct.

You will submit ONE COPY of the design at the beginning of class on the due date. Be sure to bring a 2nd copy of your design package to use in a design "WALK-THROUGH" in class on the due date.

IF YOU NEED TO SEE YOUR INSTRUCTOR FOR HELP ON THE DESIGN, bring the specifications, as much of the design as you've been able to complete on your own, and your class notes with you.

Part Two: The C++ PROGRAM.

This is due TWO WEEKS after receiving the assignment (that is, ONE WEEK after the design is due). Do not wait to receive your design package back from your instructor before proceeding on
this step. You should have your own copy of the design package to use for this phase.

The following items must be submitted in a folder at the beginning of class on the due date:

1. A single printout of your C++ program.

2. For a program that is not interactive (that is, one which reads it data from a file and writes the output to a file) you must submit a printout of the output which the program generated.

It is your responsibility to thoroughly test your program on your own data prior to submitting it. Sample data files may be provided by the instructor for use in testing your program at least two days before the program is due.

3. Your DISKETTE, labeled with your name, phone number, and "username", containing the C++ program in source code form. This should be housed in an envelope and tucked in the pocket of the folder.

Do not use this diskette for work in other classes.

Program grades will be based on the following:

i) CORRECTNESS of output;
ii) READABILITY and STYLE (includes consistent indentation, use of white space, etc);
iii) MEANINGFUL VARIABLE NAMES;

iv) MODULARITY (after the first program!); and
v) APPROPRIATE DOCUMENTATION (comment box at the top, all loops, if clauses, if-else clauses, abbreviated identifiers, etc)

IF YOU NEED TO SEE YOUR INSTRUCTOR FOR HELP ON THE PROGRAM, bring the following items with you:

i) a copy (printout) of your C++ program (most recent version);
ii) a diskette containing the program;
iii) a complete copy of your design package.

PLAN AHEAD for use of campus lab time.

Documentation standards and grading

```
/*

Programmer:     Your name here
Assignment:     Program 1
Section:        COSC 120-003
Date Due:       Oct. 8, 2004
Description:    This section must describe the program; what does the program do.

Input:  filename (physical filename)
Assumptions: A list of assumptions ex.  There must be at least one record and the data file is assumed to be correct.
*/
```

Creating a data file is done from Visual C++ just like creating a program.

Sample Design Grading Sheet (approx. values–see instructor for specific point values)

COSC 120
Design Grade Sheet
Name_____

Followed directions & completion of package 50%
- A._____ copy of problem specifications/clarifications etc. 1%
- B._____ sample input/output 5%
- C._____ input objects 2%
- D._____ output objects 2%
- E._____ other objects 2%
- F._____ structure chart 20%
  1. _____ modularity
  2. _____ clarity
- G._____ algorithm 10%
- H._____ format 8%

II. Logic of algorithm 50%
- A._____ Thought gone into the logic of the problem 40%
- B._____ Correctness 10%

Grade_____

COSC 120
Program Grade Sheet

Name_____

I. Followed directions & completion of Package 5%_____
II. Correctness of Output 50%_____
III. Readability & Style 10%_____
IV. Modularity 10%_____
V. Meaningful Object and constant names 5%_____
VI. Documentation 20%_____
- Heading _____8%
- Internal _____12%

Total _____